Available online at www.sciencedirect.com

ScienceDirect

CAAI Transactions on Intelligence Technology 1 (2016) 79–89
http://www.journals.elsevier.com/caai-transactions-on-intelligence-technology/

Original article

RPK-table based efficient algorithm for join-aggregate query on MapReduce

Zhan Li [a], Qi Feng [b], Wei Chen [c], Tengjiao Wang [a,*]

[a] *Peking University, China*
[b] *Natural Science Foundation of China, China*
[c] *The Chinese University of Hong Kong, Hong Kong, China*

Available online 28 May 2016

Abstract

Join-aggregate is an important and widely used operation in database system. However, it is time-consuming to process join-aggregate query in big data environment, especially on MapReduce framework. The main bottlenecks contain two aspects: lots of I/O caused by temporary data and heavy communication overhead between different data nodes during query processing. To overcome such disadvantages, we design a data structure called Reference Primary Key table (RPK-table) which stores the relationship of primary key and foreign key between tables. Based on this structure, we propose an improved algorithm on MapReduce framework for join-aggregate query. Experiments on TPC-H dataset demonstrate that our algorithm outperforms existing methods in terms of communication cost and query response time.

Keywords: Join-aggregate query; MapReduce; Query optimization; RPK-table; Communication cost

1. Introduction

Recently, big data attracts more and more attention. Join-aggregate query which returns aggregate information on the join of several tables is widely used in big data analysis. For instance, many TPC-H[1] queries contain joinaggregate operation for performance evaluation. However, it is time-consuming to run join-aggregate query in existing systems like Hive [1] and Pig [2].

Distributed system has been proven powerful for large-scale datasets analysis. MapReduce [3] is an important parallel computing framework in distributed system. It has been studied and applied widely in academia and industry. However, both the join and aggregate operation processed on MapReduce are time-consuming [4,5]. The main performance bottlenecks come from the following aspects: reading and writing lots of temporary data on disk and heavy communication overhead between different data nodes. The overall response time increases when the data size scales up.

Consider of the following query in TPC-H:

SELECT c.name, COUNT(*) FROM Customer c, Orders o WHERE c.custkey = o.custkey GROUP BY c.name

This SQL query uses TPC-H benchmark schema and computes the number of orders that every customer takes. As the join attribute and aggregate attribute are not the same, we can't execute join operation and aggregate operation in one single MapReduce job. Traditional approach to execute this query need two MapReduce jobs. First job loads two table, exchanges tuples between different data nodes, performs the join operation on the two datasets and then writes the joined result on disk. Then the second job loads the temporary joined result and computes the final aggregate results. The overall computation is time-consuming due to the heavy communication cost and I/O cost. When queries contain multiple joins

* Corresponding author. Internet Research and Engineering Center, School of Electronic and Computer Engineering, Peking University, China. Tel.: +86 755 26035225.
E-mail address: tjwang@pku.edu.cn (T. Wang).
Peer review under responsibility of Chongqing University of Technology.
[1] www.tpc.org/tpch.

http://dx.doi.org/10.1016/j.trit.2016.03.008

on different tables, the query performance will continue to decrease.

To overcome the above drawbacks, we design a data structure called Reference Primary Key table (RPK-table). In RPK-table we store the relationship of primary key and foreign key between tables. This structure is independent of remote data distribution and movement in environment. We also propose a new algorithm for join-aggregate query execution on MapReduce. We only need one single MapReduce job for join-aggregate query. The major performance disadvantages, communication cost and I/O cost, is reduced in our algorithm with the help of RPK-table. Experiments in Section 4 validate the effectiveness of our proposed algorithm in improving the query response time.

The rest of the paper is organized as follows. We review the related work in Section 2. In Section 3, we describe our proposed structure RPK-table and explain our algorithm processed on MapReduce framework. Section 4 shows the performance of our algorithm. Finally we state the conclusion in Section 5.

2. Related work

Both aggregate query and join query have been studied in many recent research works. We analyze some related works which optimize query processing in different situations. These fall into the following broad categories:

- Optimization method for performance improvement in traditional database. Previous works such as [6,7] propose several methods to improve the query performance in traditional database. Their methods like concurrent execution of multiple queries, indexing techniques and physical database design are hard to be implemented and maintained in distributed environment. They also do not consider communication cost during query execution. Join Partition Method (JPM) and Aggregate Partition Method (APM) [8] are two parallel processing methods for join-aggregate query. The JPM method need to process aggregate and join operation separately. The APM method need to broadcast one whole table into all processors. These two methods are sub-optimal on MapReduce framework.
- Modifying or extending MapReduce framework to improve performance. Some researchers [11,12] extend map-reduce model to process join aggregate in one job. Some researchers [13–15] aim to improve the performance by building middleware or cache structure on top of MapReduce framework. These are optimizations specifically on join strategies and they need to modify the core framework of MapReduce. Our proposed algorithm, on the other hand, is more general and can improve the efficiency of join-aggregate operator.
- Pre-computing query results. Some analysis systems store huge amount of data for decision-making. The time interval of data updating in these systems is long enough. Thus some query results can be pre-computed ahead and stored on the disk [16,17]. However, as new data is generated more and more rapidly, data updating becomes more and more frequently. Re-computing query results when data updates becomes complicate and time-consuming. Also this approach cannot assure the efficiency of adhoc query processing for data-intensive application. We focus on improving join-aggregate computation without pre-computation techniques.
- Approximate query processing. Some researchers use approximation method to decrease query response time. These approaches contain sample method [21] and online computation [18–20]. They return an approximate result with a certain error bound or guarantee for each query. The response time will decrease a lot but they do not provide an exact result. Furthermore, its requirement for random data retrieval makes it difficult to be performed on distributed system. The goal of our work is to design an algorithm which improves the performance of join-aggregate queries without sacrificing accuracy.

3. Our approach

In distributed system the whole dataset is partitioned and located in different nodes depending on the availability of storage resources. During join-aggregate query execution, each worker accesses data splits from different nodes and then performs join operation on primary and foreign key attributes.

After that the required attribute values are filtered and the aggregate results are computed. When the data size is huge, the execution may become complex. Lots of temporal data need to be written and read in disk and the communication overhead between different nodes may increase heavily. This will result in increasing query processing time and decreasing system performance. To achieve better performance, we first design a new data structure which minimizes the storage cost. Then we propose an optimization join-aggregate query processing algorithm on MapReduce with the help of our structure.

3.1. Data structure RPK-table

In this sub-section, we describe the Reference Primary Key table. Then we explain the reason for designing it and the way to store it.

First we give our definitions here.

Definition 1. Table which contains the primary key in table relationship between primary and foreign key is defined as **target table**.

Definition 2. Table which contains the foreign key in table relationship between primary and foreign key is defined as reference table.

Since join-aggregate queries often perform equi-join on tables in their primary and foreign keys, we design a structure called RPK-table. In RPK-table, we store the

relationship between primary and foreign key of reference tables. For each target table we create a RPK-table for its primary key. Thus the mapping information of primary and foreign key between target table and reference table can be directly access from the RPK-table. We store target table's primary key attribute in the first column and subsequent columns contain reference tables'primary key attribute. Those records stored in the subsequent columns have the same foreign key values which are equal to the mapping records' primary key value in first column. When the original datasets update, we only need to update corresponding portal of RPK-table. For each tuple in RPKtable, it is updated only when the corresponding records in target table and reference table are updated.

We use the TPC-H benchmark to study and analyze the performance of our work. TPC-H [22] is a widely used decision support benchmark. It is designed to evaluate the functionalities of business analysis applications. In TPC-H schema, we need to create RPK-table for each target table. RPK-tables for Customer table and Nation table in TPC-H schema are in Table 1 and Table 2.

In Table 1, the target table is Customer and the reference table is Orders. The first column stores sorted primary key value, custkey, for every record in Customer table. The subsequent column contains the mapping records' primary key in table Orders. The record1's primary key is equal to 001. For Orders table, these records whose primary key is 002 or 003 have the same value (001) in their foreign key, custkey. These records will be joined with record1 during join processing.

Table 1
RPK-Table for customer table and orders table.

(a) Customer table

	custkey	name	…
record1	001	Logan	…
record2	002	Scott	…
record3	003	Charles	…
…	…	…	…
record25	025	Erik	…
…	…	…	…

(b) Orders table

	orderkey	custkey	…
record1	001	023	…
record2	002	001	…
record3	003	001	…
…	…	…	…
record25	025	094	…
…	…	…	…

(c) RPK-table for Customer

	c.custkey	o.orderkey	…
record1	001	002,023	…
record2	002	008,012,003	…
record3	003	005	…
…	…	…	…
record25	025	011,022,033	…
…	…	…	…

Table 2
RPK-table for nation table customer table and supplier table.

(a) Nation table

	nationkey	name	…
record1	001	Mulgore	…
record2	002	Stormwind	…
record3	003	Orgrimmar	…
…	…	…	…
record25	025	Ironforge	…
…	…	…	…

(b) Customer table

	custkey	nationkey	…
record1	001	001	…
record2	002	042	…
record3	003	002	…
…	…	…	…
record25	025	091	…
…	…	…	…

(c) Supplier table

	suppkey	nationkey	…
record1	001	015	…
record2	002	001	…
record3	003	025	…
…	…	…	…
record25	025	002	…
…	…	…	…

(d) RPK-table for Nation table

	n.nationkey	c.custkey	s.suppkey
record1	001	001,012	002
record2	002	003	004,025
record3	003	014,015,016,018	023
…	…	…	…
record25	025	081,005	003
…	…	…	…

In Table 2, the target table is Nation and the reference tables are Customer and Supplier. The first column stores sorted primary key value, nationkey, for every record in Nation table. Two subsequent columns contain the mapping records' primary key in table Customer and Supplier. The record1's primary key is equal to 001. For Customer table, these records whose primary key is 001 or 012 have the same value (001) in their foreign key, nationkey. These records will be joined with record1 during join processing.

3.2. Our algorithm on MapReduce

In distributed systems datasets are horizontally partitioned into data splits by primary key. Then each split is distributed to different data nodes and contains consecutive records sorted by primary key. Each data node contains several data splits based on resource available. Under such situation, we generate RPK-table for each target table in dataset. Then we horizontally partition them using the same partition keys which are used to partition original table. After that we distribute each partition into the same node which contains the corresponding data splits. With the information stored in RPK-table, we can

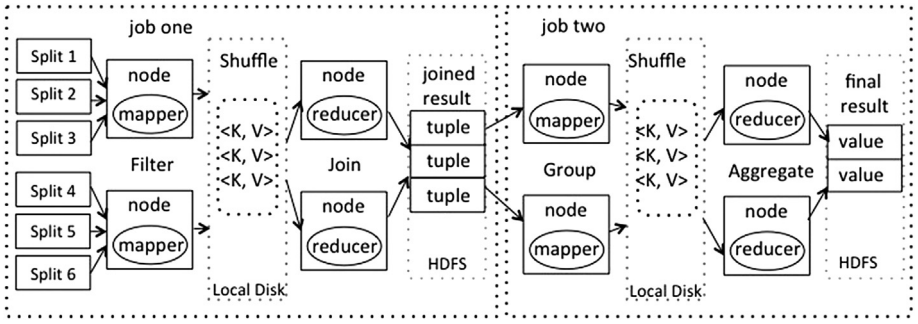

Fig. 1. Traditional approach to process query 1.

process join operation in Mapper without scanning reference table. Then the aggregate operation is processed in each Reducer separately. We describe the detailed procedure of our algorithm on MapReduce in thecoming sections. We focus on MapReduce framework since MapReduce has been studied and applied widely in academia and industry. Nevertheless, the algorithm is not only exclusive to MapReduce framework. It can be applied in different systems.

3.2.1. Predication and aggregate operation on target table

When filter operation and aggregate operation are on attributes in target table or the aggregate operator is COUNT, we can eliminate the scanning of reference table. The relationship between target table and reference table can be accessed from the RPK-table instead of original table. As the data splits and corresponding RPK-table partitions are all sorted by primary key, we scan RPK-table and target table to get the joined result. As the data splits and their corresponding RPK-table partitions are stored in the same data nodes, we can eliminate the communication cost between different nodes. We only need to access the required part of attributes whenever necessary.

Take query 1 for example, we have mentioned that traditional method to process this query need two MapReduce jobs. The first job computes the joined results and the second one generates final aggregate result. Fig. 1 illustrates the whole approach for processing query 1. These two jobs run separately. The temporary data need to be written and read in disk between two jobs. And the whole computation need to transfer many records between data nodes to compute the joined result. Thus the overall cost is expensive.

When using RPK-table, we only need one MapReduce job to finish the computation. We can eliminate the first MapReduce job and compute the aggregate and join operation in one MapReduce job. We should mention that having clause can be converted into ”where” clause constraints by rewriting the query [9], thus we do not consider having clause in our algorithm.

Fig. 2 presents the MapReduce job generated by our algorithm. As this query tries to find the number of orders that

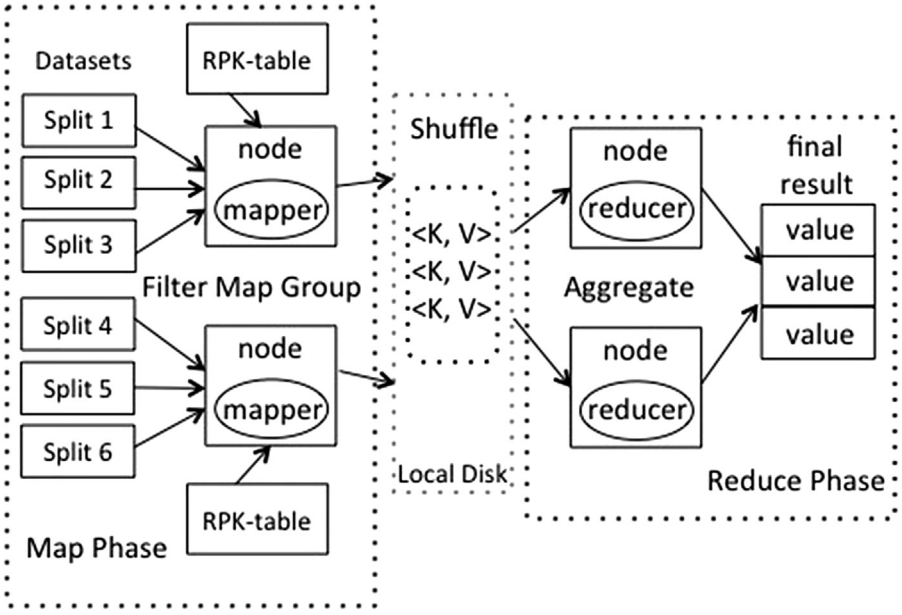

Fig. 2. Our approach to process query 1.

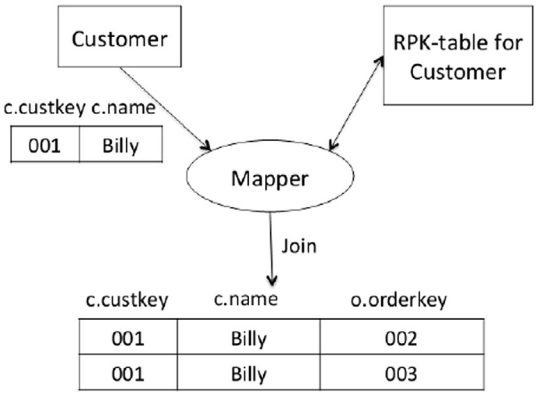

Fig. 3. Procession in Mapper.

each customer takes, we only need to count the number of rows in Orders table that belong to each customer. Thus the scanning of Orders table can be eliminated. We can get the number of reference records by scanning corresponding RPK-table for Customer table. As shown in Fig. 3, The join operation is processed in Mapper by scanning each split of Customer table and corresponding RPK-table partition.

These RPK-tables are much smaller compared to Orders table. At the same time, the data splits and their associated RPK-tables are stored locally. Thus the communication cost only occurs in shuffle phase. In shuffle phase we partition the joined results, which contain attributes, c.name and o.custkey, in this case and then transfer them to each Reducer. The partition key is customer name in this query. Finally, we compute the aggregate result using hash table [10] in Reducer as in Fig. 4.

3.2.2. Predication and aggregate operation on reference table

When filter operation and aggregate operation are on attributes in reference table, we need original data records to compute the final results. Suppose we have MAX (orderdate) instead of COUNT (*) in query 1, we need the real values of attribute orderdate to compute the aggregate result. In this case we can't eliminate accessing attributes from reference table. However, we can use RPK-table to access only useful portion of attributes by pushing down the group-by clause in the query plan. Hence we use only one MapReduce job to complete the whole query computation.

In Mapper we still scan RPK-table instead of reference table. We use target table and RPK-table to compute the joined results. In Reducer, we need to fetch the required attributes (orderdate) to compute final results. As we have the primary key of each record in joined result, we can access the whole record or only necessary part of records from remote nodes. The data movements between nodes are still smaller than common computation.

The detail query processing algorithm on MapReduce is showed below.

Algorithm 1 Join-aggregate Query Processing Algorithm

```
1: Input: Query Q, Target Table T, RPK-tables RT
2: Output: Result of Q
3: function map():
4: for record r of T do
5: access those records whose primary key is reference by r in RPK-table
6: if there is only predication and aggregate operation on target table then
7: store the records' primary keys
8: else
9: fetch the required attributes by primary key
10: apply predicate on those attributes
11: store the required attributes in order
12: end if
13: emit (group-by key, required values)
14: end for
15:
16: shuffle on group-by key
17:
18: function reduce (key, list < r >)
19: fetch other attributes if necessary
20: communicate with other nodes if necessary
21: compute aggregate result using hash table
22: compute final result using part aggregate values
```

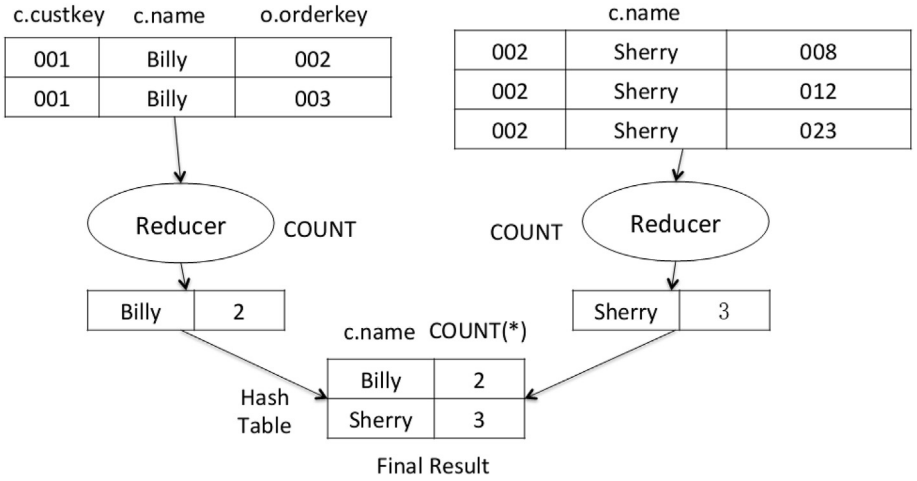

Fig. 4. Procession in Reducer.

```
SELECT N.name, SUM(O.totalprice)
FROM ORDERS O, CUSTOMER C, NATION N
WHERE N.nationkey = C.nationkey
        and C.custkey = O.custkey
        and O.orderdate like '1995%'
GROUP BY N.NAME
```

Fig. 5. Complex query contains multiple join operations in TPC-H.

As for more complex queries the query shown in Fig. 5, we can use ChainMapper to process this query, which allows to use multiple Mappers within a single MapReduce job.

We have two join operations on three tables. Thus we need two mappers and one reducer to finish the job. In the first Mapper we begin with scanning the table which is notperformed as reference table during the whole execution. Then each subsequent Mapper processes one join operation with the help of RPK-table and the previous result. We finally generate the query result in reducer. Hence we need to transfer data between each Mapper, but we only need to transfer required attributes and primary key values. When we need other attributes to complete computation, we can access them from remote nodes by the records' primary keys. The communication cost is decreased as much as possible. For queries containing more than one group-by keys, we change the order of attributes in group-by clause corresponding to theorder of table processed in our job. Finally we can compute the aggregate result in Reducer using hash table.

4. Experiments

In this section we conduct two experiments to validate the effectiveness of our proposed structure and algorithm. We compare our algorithm with Hive and Pig through join aggregate queries in TPC-H benchmark.

4.1. Experiments settings

We run our experiments on a cluster with 20 nodes. We use 19 nodes as slaves and one node as master. The detailed systems are shown in Table 3. We use Hadoop-1.2.1, Hive-0.10.0 and Pig-0.7.0 in our experiments.

For our experiments we generate 500 GB data using TPCH tool "dbgen" and then distribute it in our cluster. Then the RPK-tables for each target table in our TPC-H datasets are generated. The structure of TPC-H datasets are shown in Fig. 7. We only need scan the target table and reference table once for each RPK-table. It only takes several minutes to generate each RPK-table. After that we horizontally partition them using the same partition keys that are used to partition each target table. Then we distribute each RPK-table partition into data nodes which contain corresponding data splits.

We use five different queries for our performance analysis. The used queries are Q3, Q10, Q12, Q14, Q19 in TPC-H benchmark which are shown in Table 4. The differences among those queries are the number of join operations, the type of aggregate operation, the table size and the filtration rate.

4.2. Storage cost of RPK-table

We present the size of RPK-table with different data size in Fig. 8. The storage cost of our structure is only about 10% of the original datasets. As the structure storage cost is small

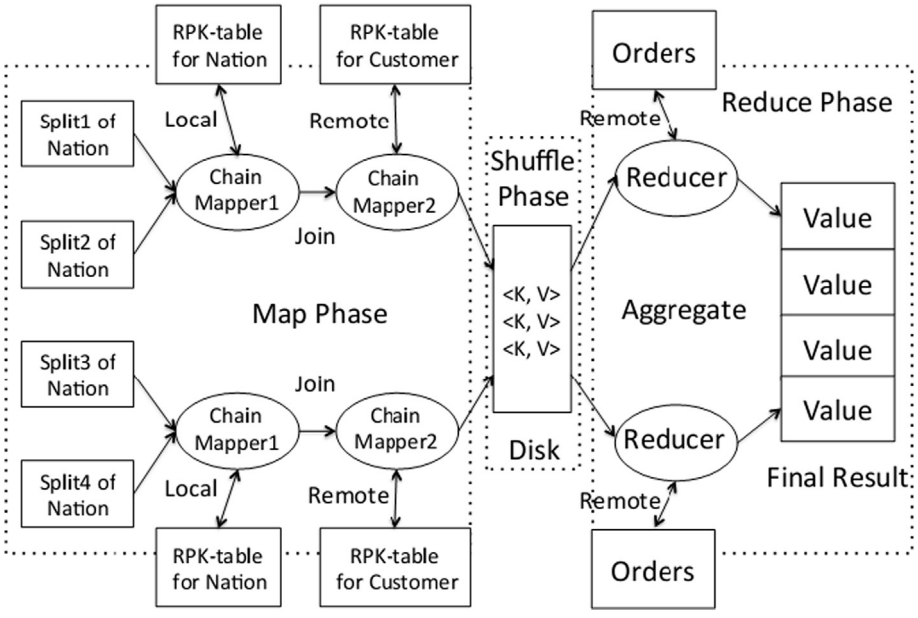

Fig. 6. ChainMapper for complex queries.

Table 3
Systems settings.

	CPU	Operator System	Memory	Disk
master	4 Quad- Core AMD OpteronTM Processor 2378 2.4 GHz	Ubuntu 12.04 LTS	32 GB	1 TB
slave	4 Quad- Core AMD OpteronTM Processor 2378 2.4 GHz	Ubuntu 12.04 LTS	32 GB	500 GB

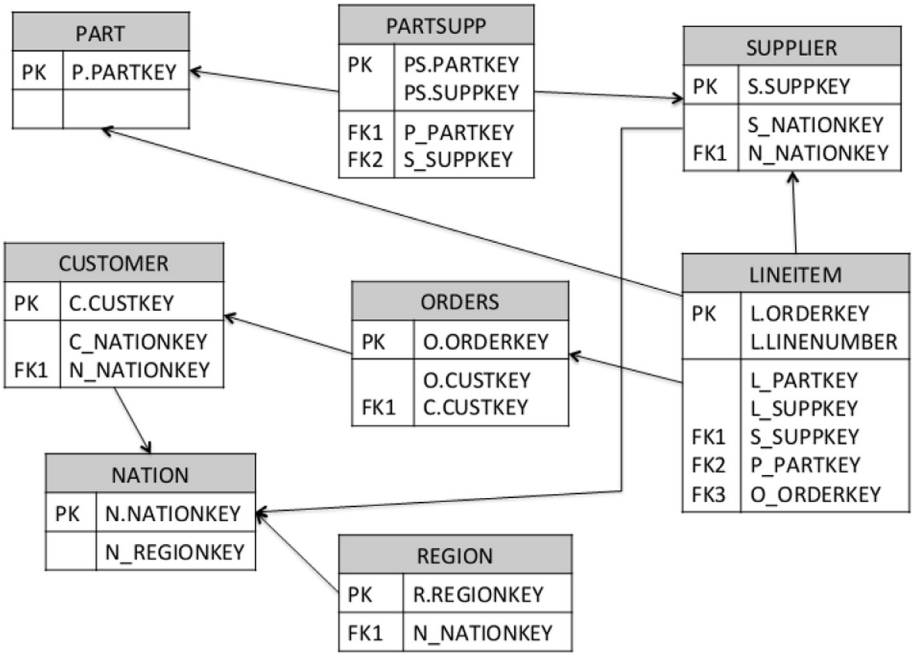

Fig. 7. The structure of TPC-H datasets.

enough, we load the RPK-tables into main memory and thus can use the structure directly.

4.3. Inter-node communication cost

To compare the inter-node communication cost among our proposed algorithm, JPM method and APM method, we compute the size of data exchanged during query processing. The results are presented in Fig. 9.

In our algorithm most of join operations are done locally. In shuffle phase we only transfer primary key values and required attributes for aggregate computation in Reducer. We can know from Fig. 9 that the total communication cost in our approach is much less than other methods.

Table 4
Queries in experiment.

Query	Function	datasheet
Q3	Shipping Priority Query	Customer, Orders, Lineitem
Q10	Returned Item Reporting Query	Customer, Orders, Lineitem, Nation
Q12	Shipping Modes and Order Priority Query	Orders, Lineitem
Q14	Promotion Effect Query	Lineitem, Part
Q19	Discounted Revenue Query	Lineitem, Part

4.4. Update cost of RPK-table

When the records in dataset are updated, changed or deleted, we need to update the related data in RPK-table. So this could take more time when date updated. Thus we compare the time costs when data updated with and without RPK-table. The results are present in Fig. 10.

We can know that the time overhead are similar in two situations. It does not cause too much additional time overhead when using RPK-table.

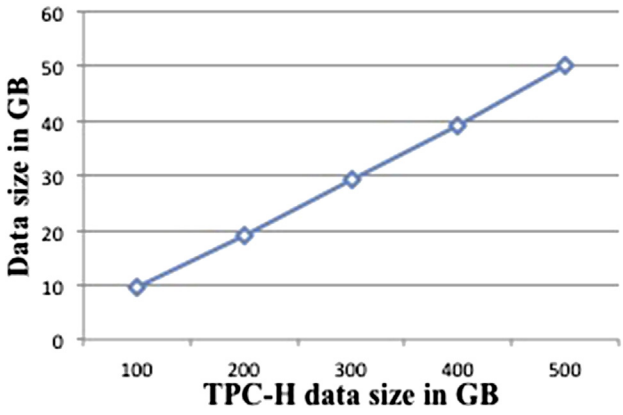

Fig. 8. RPK-table size in GB.

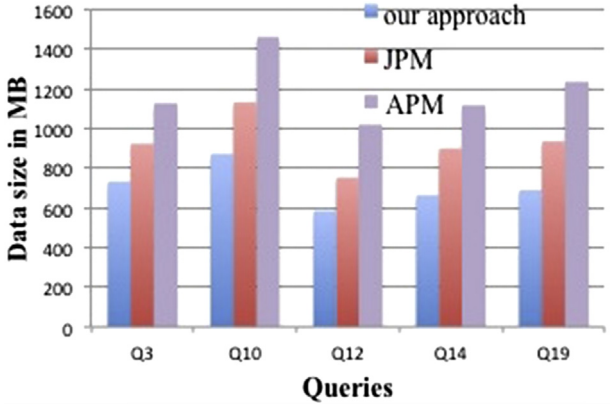

Fig. 9. Exchanged data size in MB.

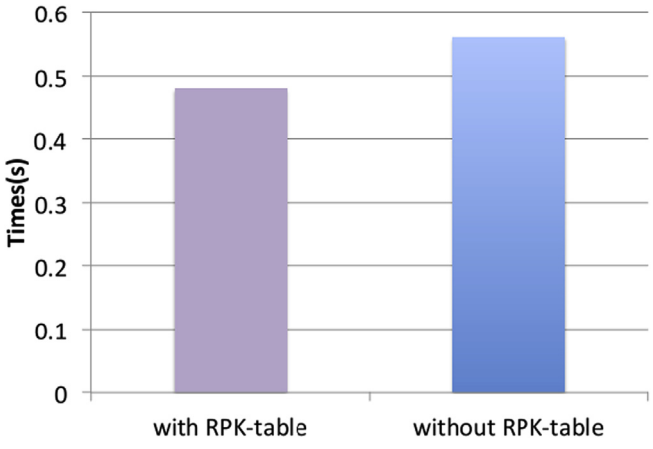

Fig. 10. Data update cost.

4.5. Query execution

We compare the performance of our algorithm with Hive, Pig, JPM method and APM method. We present the comparison of time consumed by these five queries in Fig. 11–15.

Hive, Pig and JPM method all use more than one MapReduce job to complete the whole query execution. They need one MapReduce job for each join operation and one MapReduce job for aggregate operation. Each node has to communicate partial result, write and read temporary data between consecutive jobs. The APM method only need one job for Q10, Q12, Q14 and Q19. But it need to broadcast and scan the table which does not contain group-by attribute in each Reducer. And it still need more than one job to process queries like Q3 in which the group-by attributes are from more than one table. Our algorithm only need one MapReduce job to complete each query and eliminate communicating original records whenever possible. We decrease communication cost between different nodes and reduce computation cost in each job. As a result, the overall time consumed is reduced. Based on the results illustrated in Fig. 6, our algorithm is faster than other approaches in terms of the total execution time. The main reasons for this improvement include less data read and written on HDFS, fewer job spawned, less job setup time, less checkpointing cost and communication overhead.

We can conclude that our algorithm works well for join-aggregate query on MapReduce. As we mention above, our approach is not only suit for MapReduce framework. In other cloud architectures, we can still generate RPK-tables for each target table and use it to eliminate the scanning of original datasets.

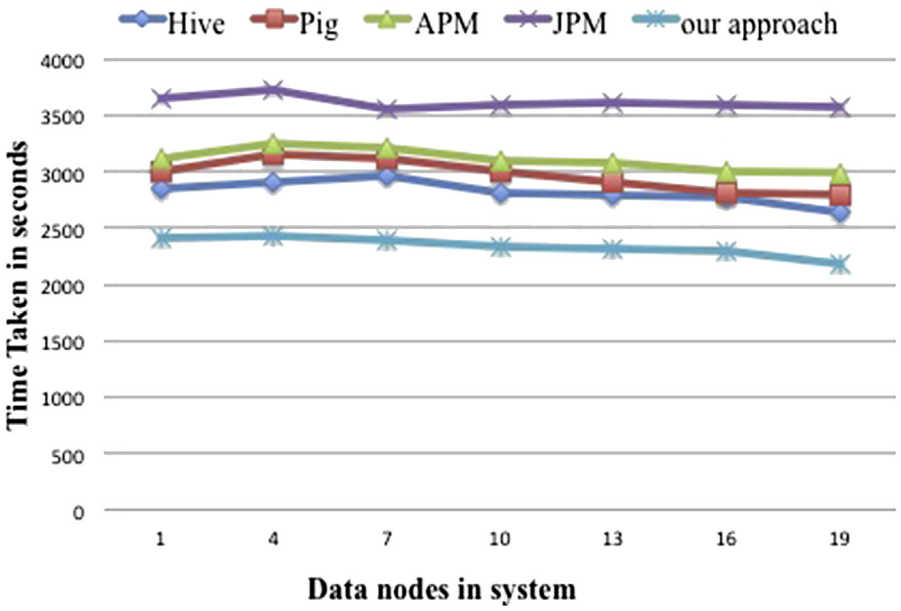

Fig. 11. Query 3.

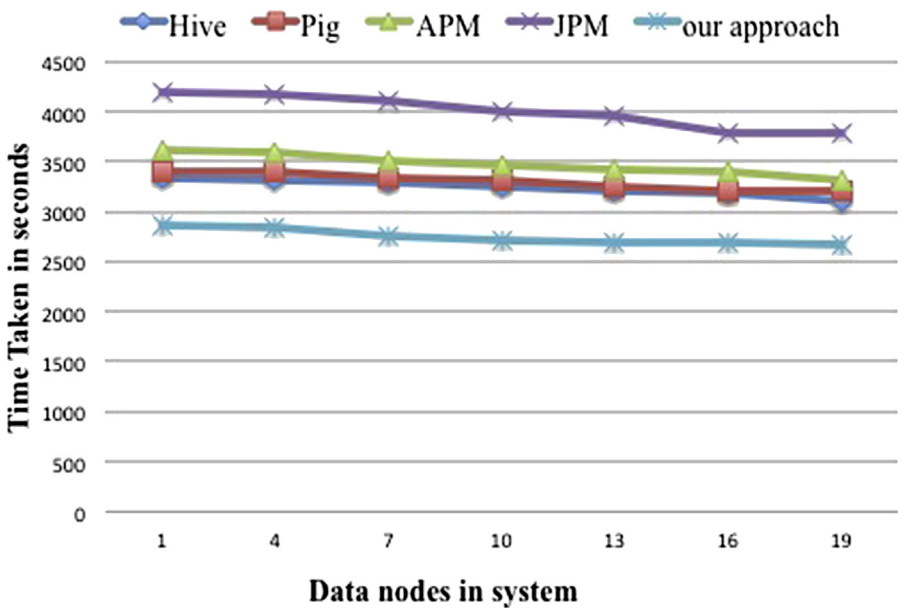

Fig. 12. Query 10.

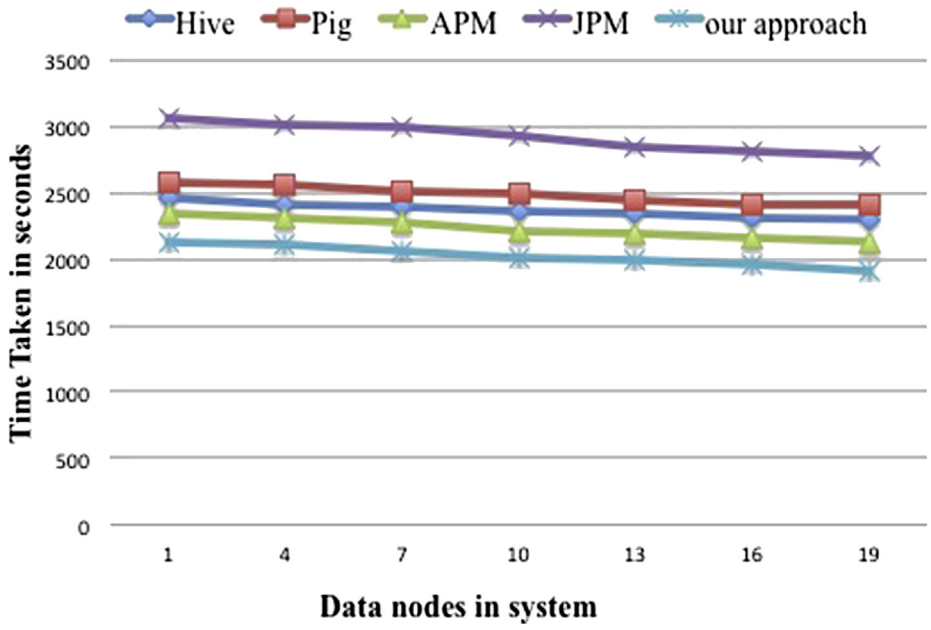

Fig. 13. Query 12.

5. Conclusion

In this paper, we study and analyze join-aggregate query processed in distributed environment. We design a data structure with low space overhead, called RPK-table. The structure stores information of relationship between target and reference tables on their primary and foreign key. Then we propose our algorithm on MapReduce to process aggregate queries containing join operation on large-scale data. Our approach improves the performance of query execution by reducing communication overhead, decreasing job setup time and disk I/Os. The experiments confirm that our algorithm has a better performance than existing approaches.

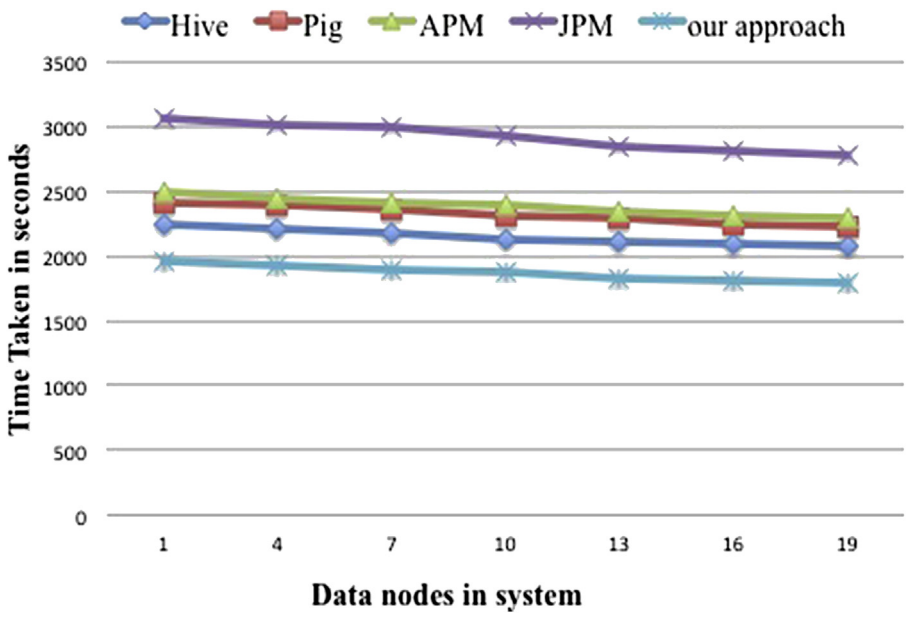

Fig. 14. Query 14.

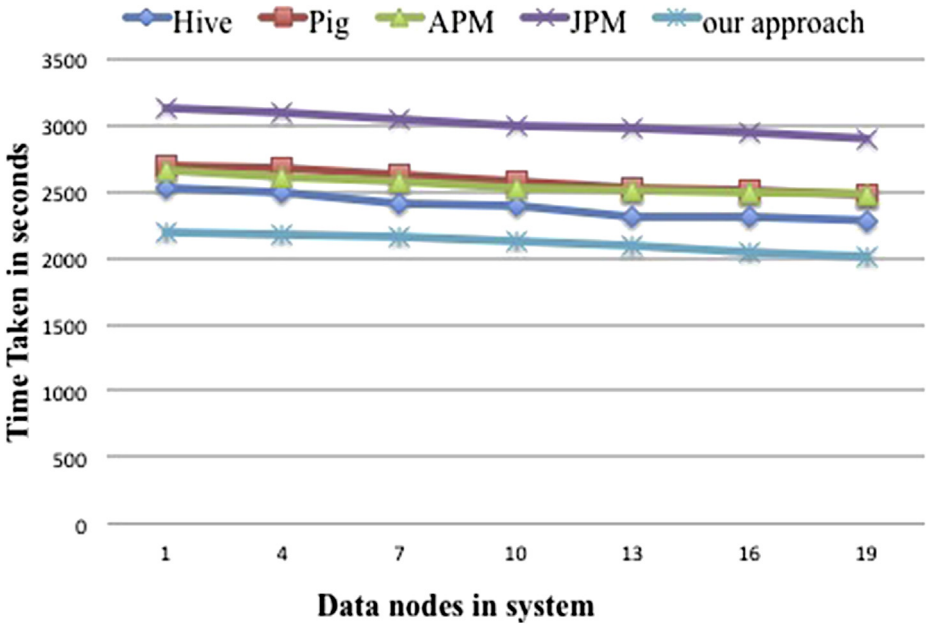

Fig. 15. Query 19.

References

[1] A. Thusoo, J.S. Sarma, N. Jain, et al., Proc. VLDB Endow. 2 (2) (2009) 1626–1629.
[2] C. Olston, B. Reed, U. Srivastava, et al., Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, 2008, pp. 1099–1110.
[3] J. Dean, S. Ghemawat, Commun. ACM 51 (1) (2008) 107–113.
[4] K.H. Lee, Y.J. Lee, H. Choi, et al., AcM sIGMoD Rec. 40 (4) (2012) 11–20.
[5] C. Doulkeridis, K. Nrvg, VLDB J. 23 (3) (2014) 355–380.
[6] M. Muralikrishna, VLDB 92 (1992) 91–102.
[7] M.O. Akinde, M.H. Bohlen, Efficient computation of subqueries in complex OLAP, in: Data Engineering, 2003. Proceedings. 19th International Conference on. IEEE, 2003, pp. 163–174.
[8] D. Taniar, R.B.N. Tan, C.H.C. Leung, et al., Inf. Sci. 168 (1) (2004) 25–50.
[9] W.P. Yan, P.B. Larson, VLDB 95 (1995) 345–357.
[10] Y. Yu, P.K. Gunda, M. Isard, Distributed aggregation for data-parallel computing: interfaces and implementations, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 247–260.
[11] S.Y. Chen, P.C. Chen, An efficient join query processing based on MJR framework, in: Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on. IEEE, 2012, pp. 698–703.

[12] H. Yang, A. Dasdan, R.L. Hsiao, et al., Map-reduce-merge: simplified relational data processing on large clusters, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, 2007, pp. 1029−1040.
[13] S. Zhang, J. Han, Z. Liu, et al., Accelerating MapReduce with distributed memory cache, in: Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on. IEEE, 2009, pp. 472−478.
[14] M. Zaharia, A. Konwinski, A.D. Joseph, et al., OSDI 8 (4) (2008) 7.
[15] D. Peng, K. Duan, L. Xie, Int. J. Database Theory & Appl. 6 (1) (2013).
[16] D. Xin, J. Han, X. Li, et al., Star-cubing: computing iceberg cubes by top-down and bottom-up integration, in: Proceedings of the 29th International Conference on Very Large Data Bases, Vol. 29, VLDB Endowment, 2003, pp. 476−487.
[17] M.C. Hung, M.L. Huang, D.L. Yang, et al., Inf. Sci. 177 (6) (2007) 1333−1348.
[18] S. Agarwal, B. Mozafari, A. Panda, et al., BlinkDB: queries with bounded errors and bounded response times on very large data, in: Proceedings of the 8th ACM European Conference on Computer Systems, ACM, 2013, pp. 29−42.
[19] N. Pansare, V.R. Borkar, C. Jermaine, et al., Proc. VLDB Endow. 4 (11) (2011) 1135−1145.
[20] X. Han, J. Li, H. Gao, Inf. Sci. 278 (2014) 773−792.
[21] N. Laptev, K. Zeng, C. Zaniolo, Proc. VLDB Endow. 5 (10) (2012) 1028−1039.
[22] TPC-H benchmark. http://www.tpc.org/tpch/spec/tpch2.14.4.pdf.