Join Query Processing in MapReduce Environment

Anwar Shaikh and Rajni Jindal

Dept. of Computer Engineering,
Delhi Technological University, Delhi, India
anwardshaikh@gmail.com, rajnijindal@dce.ac.in

Abstract. MapReduce is a framework for processing large data sets, where straightforward computations are performed by hundreds of machines on large input data. Data could be stored and retrieved using structured queries. Join queries are most frequently used and importatnt. So its crucial to find out efficient join processing techniques. This paper provides overview of join query processing techniques & proposes a strategy to find out best suitable join processing algorithm.

Keywords: MapReduce, Join processing, Hadoop, Multiway Join.

1 Introduction

MapReduce was proposed by Google [1]. Many complex tasks such as parallelism, fault tolerance, data distribution and load balancing are hidden from the user; thus making it simple to use. Tasks are performed in two phases, Map and Reduce. Input in the form of key/value pairs is processed by Map function to produce intermediate key/value pairs; these intermediate values with same keys are merged together by Reduce function to form smaller set of values as output.

map(InputKey, InputValue) → **list** (IntermediateKey, intermediateValue)
reduce(IntermediateKey, **list**(intermediateValue)) → **list**(intermediateValue)

Map and Reduce function are specified by the user, but the execution of these functions in the distributed environment is transparent to the user. Hadoop is open source implementation of MapReduce [2], built on top of Hadoop Distributed File System (HDFS) which could handle petabytes of data [10]. Data blocks are replicated over more than one location over the cluster to increase the availability.

2 Related Work

A framework Map-Reduce-Merge[8] was designed to improve join processing, it included one more stage called Merge to join tuples from multiple relations. Join performance could be improved by indexes; Hadoop++[6] used Trojan Join and Trojan Index to improve join execution. Methods described in this paper could be applied when data is organized in Row-wise manner; [9] described join optimization algorithms for column-wise data. Authors in [7] designed a query optimizer for Hadoop.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2012

3 Join Processing

Join algorithms used by Conventional DBMS and MapReduce are different, because join execution in MapReduce uses Map and Reduce functions to get results. This section describes various join processing algorithms for MapReduce environment.

3.1 Repartition Join

Repartition Join[5] is a default join mechanism in Hadoop[3], implemented as a single MapReduce job. A single split(block) of relation involved in join is processed by each mapper in Map phase and a set of - Join key (k1), tuple (t1), relation name (R) – {k1, t1, R} is produced as output. R is used as a tag to identify the relation to which a particular tuple belongs.

Output of Map phase is sorted and partitioned based on join key. Records with different join keys are distributed to each reducer. Tags attached to the tuples are removed and tuples are separated to form two different relations by Reducer. Final output is produced by performing cross join between these two relations.

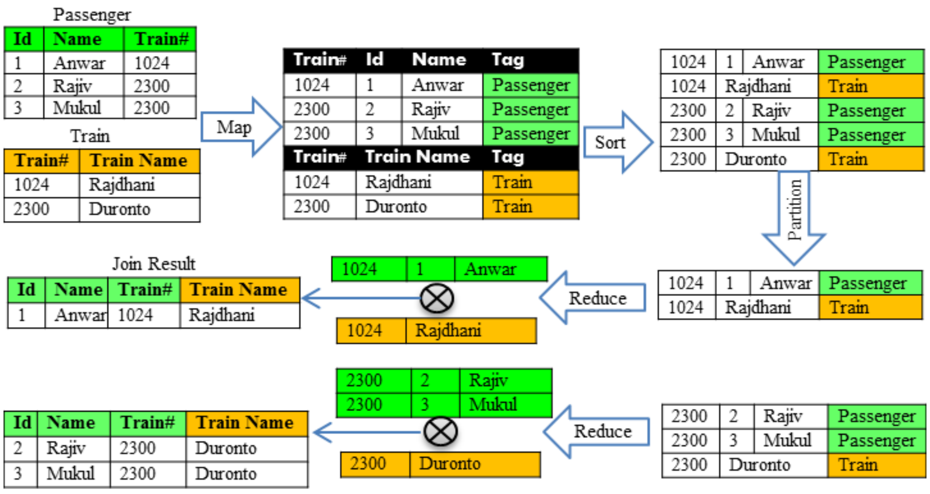

Passenger
Id | Name | Train#
1 | Anwar | 1024
2 | Rajiv | 2300
3 | Mukul | 2300
Train
Train# | Train Name
1024 | Rajdhani
2300 | Duronto
Map
Train# | Id | Name | Tag
1024 | 1 | Anwar | Passenger
2300 | 2 | Rajiv | Passenger
2300 | 3 | Mukul | Passenger
Train# | Train Name | Tag
1024 | Rajdhani | Train
2300 | Duronto | Train
Sort
1024 | 1 | Anwar | Passenger
1024 | Rajdhani | Train
2300 | 2 | Rajiv | Passenger
2300 | 3 | Mukul | Passenger
2300 | Duronto | Train
Partition
1024 | 1 | Anwar | Passenger
1024 | Rajdhani | Train
Reduce
1024 | 1 | Anwar
1024 | Rajdhani
Join Result
Id | Name | Train# | Train Name
1 | Anwar | 1024 | Rajdhani
2300 | 2 | Rajiv | Passenger
2300 | 3 | Mukul | Passenger
2300 | Duronto | Train
Reduce
2300 | 2 | Rajiv
2300 | 3 | Mukul
2300 | Duronto
Id | Name | Train# | Train Name
2 | Rajiv | 2300 | Duronto
3 | Mukul | 2300 | Duronto

Fig. 1. Repartition Join

Default Hadoop Join mechanism is not efficient because of following reasons –

1. Sorting and movement of all tuples between map and reduce phases.
2. For a popular key many tuples are sent to one reducer and buffering is required.
3. Tagging of tuples involves minor overhead in Map phase.

Improved Repartition Join was suggested by authors of [5] where the output of map phase was adjusted such that tuples of smaller relation appeared before tuples of larger relation and generation of join result needed buffering the tuples of smaller relation and streaming the tuples of larger relation.

3.2 Broadcast Join (Naïve Asymmetric Join)

Broadcast join described by [5] is similar to Naïve Asymmetric join in [3]. Broadcast join is asymmetric because both relations are treated differently. When relations R1 has very less number of tuples compared to R2, then R1 is copied to all mapper nodes using Distributed Cache Mechanism provided by HDFS. A Hash table is built at mapper such that Join attribute act as Hash key and the tuple of R1 act as Value.

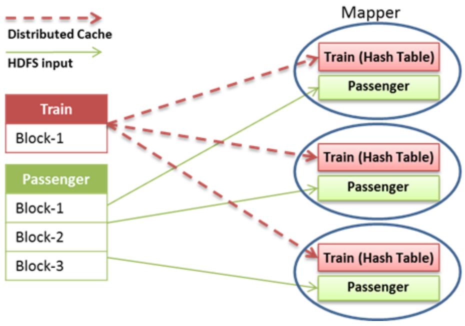


Fig. 2. Broadcast Join

A split of R2 is assigned to each Mapper. Tuples of R2 are scanned sequentially and join key attribute of R2 is hashed to find matching tuples of R1 from hash table. Note that only one Map phase is required for Broadcast join, and transmission of only the smaller relation over network reduces the bandwidth requirement.

3.3 Optimized Broadcast Join

It is also termed as Optimized Naïve Asymmetric Join [3] and Semi join[5]. Sometimes many tuples might not contribute to join result. It might be costly to broadcast large relations R1 & R2. But when selectivity of R1 is less, then size of R1 could be reduced by extraction of tuples contributing to the join result by using semi join mechanism. Two MapReduce jobs are required to perform Semi join between R1 and R2 and one more Map only job needed to perform actual broadcast join. Projection of unique join attribute values from relation R2 is done in first MapReduce. These unique values are used to find matching tuples from R1 in second MapReduce job, hence size of R1 is reduced and made suitable for broadcasting.

3.4 Trojan Join

Prior knowledge of schema and join conditions could help in improvement of join performance. Trojan join [6] was designed to take advantage of this. Also Trojan indexes were created at data load time with read optimization. Implementation of Trojan Join along with Trojan index is termed as Hadoop++.

Application of same partitioning function to both relations involved in join at data load time, called as Co-partitioning is the basic idea behind Trojan join. Co-group pairs from each of two relations having similar join attribute values are kept on the same split. Availability of data from both relations with same join key value at the same split, makes execution of join possible locally at mapper node without need of shuffle and reduce phases, hence reducing the network communication. Join result is obtained by performing cross product between Data from a Co-Group in Map phase. Execution of Trojan join between relations Passenger & Train is depicted in Figure 3.

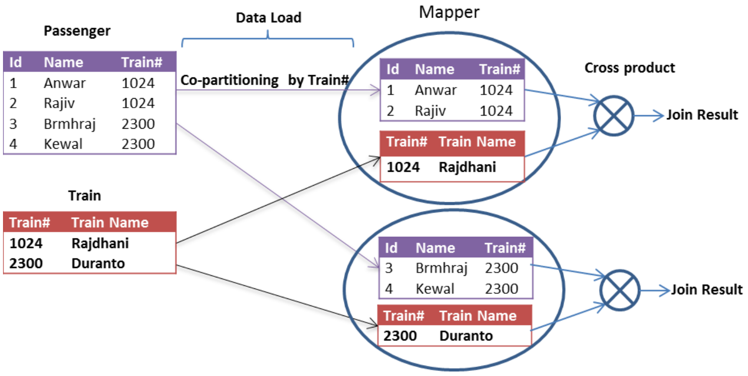
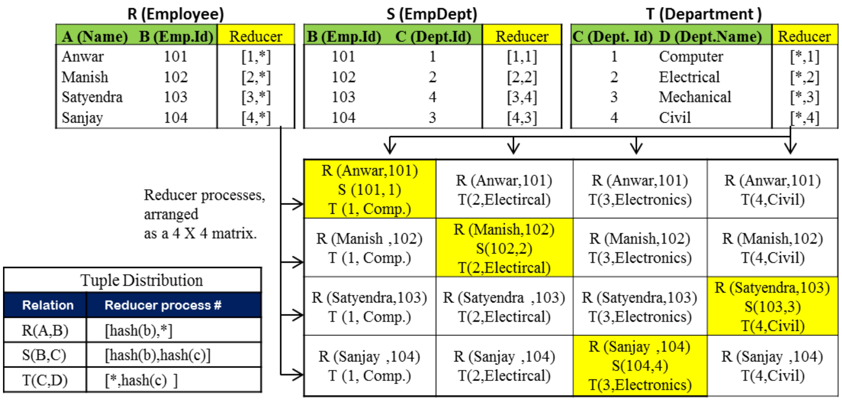


Fig. 3. Trojan Join execution

3.5 Replicated Join

Two MapReduce jobs are required to perform Natural join between three relations R(A,B), S(B,C), T(C,D) using methods described above, because only two relations can be joined at a time. Authors in [4] proposed a multiway join query processing algorithm, where three-way join can be performed as a single MapReduce operation. Tuples from relations R and T are sent to multiple reducers, communication cost might be increased, but it is acceptable because the join will be performed in the single MapReduce job. Each tuple from S is sent to single reducer only.



R (Employee)

A (Name)	B (Emp.Id)	Reducer
Anwar	101	[1,*]
Manish	102	[2,*]
Satyendra	103	[3,*]
Sanjay	104	[4,*]

S (EmpDept)

B (Emp.Id)	C (Dept.Id)	Reducer
101	1	[1,1]
102	2	[2,2]
103	4	[3,4]
104	3	[4,3]

T (Department)

C (Dept. Id)	D (Dept.Name)	Reducer
1	Computer	[*,1]
2	Electrical	[*,2]
3	Mechanical	[*,3]
4	Civil	[*,4]

Reducer processes, arranged as a 4 X 4 matrix.

R (Anwar,101) S (101, 1) T (1, Comp.)	R (Anwar,101) T(2,Electircal)	R (Anwar,101) T(3,Electronics)	R (Anwar,101) T(4,Civil)
R (Manish ,102) T (1, Comp.)	R (Manish,102) S(102,2) T(2,Electircal)	R (Manish,102) T(3,Electronics)	R (Manish,102) T(4,Civil)
R (Satyendra,103) T (1, Comp.)	R (Satyendra ,103) T(2,Electircal)	R (Satyendra,103) T(3,Electronics)	R (Satyendra,103) S(103,3) T(4,Civil)
R (Sanjay ,104) T (1, Comp.)	R (Sanjay ,104) T(2,Electircal)	R (Sanjay ,104) S(104,4) T(3,Electronics)	R (Sanjay ,104) T(4,Civil)

Tuple Distribution

Relation	Reducer process #
R(A,B)	[hash(b),*]
S(B,C)	[hash(b),hash(c)]
T(C,D)	[*,hash(c)]

Fig. 4. Replicated join

Replicated join could be used when a tuple from one relation is joined with many tuples of other relation. ‘k’ number of reduce process are selected, where k=m*m. Reducers are numbered as [i,j] where values of i and j are 1,2,..m. Tuples from R, S, & T are hashed into ‘m’ buckets by a Hash function. Tuples are sent to reducer using the hashed values of join attributes B and C. Join is performed locally at reducer as tuples from R, S, T with same join attribute value are available at reducer numbered [hash(b), hash(c)]. Distribution of tuples to the reduce processes is shown in figure 4,where k=16=4*4, final results are marked in yellow. An optimization algorithm to find minimum number of replicas/reducers needed for join execution was proposed and applied to star and chain join in [4].

4 Experiments

Experiments to evaluate performance of join processing algorithm are described in this section. Performance was evaluated for three different sized user and log relations on cluster of 6 nodes with split size of 64MB by authors of [3]. Naive Asymmetric Join took half of the time taken by Default Hadoop Join.

Optimized Broadcast Join was performed between user table and log table such that 50%, 70% and 90% log table tuples were associated to user table [3]. Results showed that time required for semi join was very less compared to actual join phase. Optimized Broadcast join performed better than Broadcast Join.

Experiments conducted in [5], on 100 node cluster showed that Improved repartition always performed better than Repartition join. And performance of Broadcast join was decreased as the number of referenced tuples and percentage of referenced tuples increased. Semi join was not observed to perform better than broadcast Join, because of high overhead of scanning entire table. Also, Scalability of Improved Repartition Join, Broadcast Join and Semi join was observed to be linear.

Performance of cascade of binary joins and three way join using replication was evaluated by authors of [4] on four node cluster, processing time taken by both approaches proved that three-way-join took less time than cascade of two way joins.

Experiments conducted on Amazon EC2 cloud showed that performance of Trojan Join was better than Hadoop [6]. Performance of Hadoop++ with split size of 1GB was better than Hadoop by factor of 20. But, for split size of 256MB performance was alike, thus increasing the split size improved the performance of Hadoop++, but reduced the fault tolerance.

5 Join Algorithm Selection Strategy

Based on the results of experiments described above, we have proposed a join algorithm selection strategy, depicted as a decision tree in Figure 5. One such strategy was proposed in [5] based on tradeoff between few join algorithms and preprocessing of data, but we have considered more number of join algorithms and assumed no preprocessing of data.

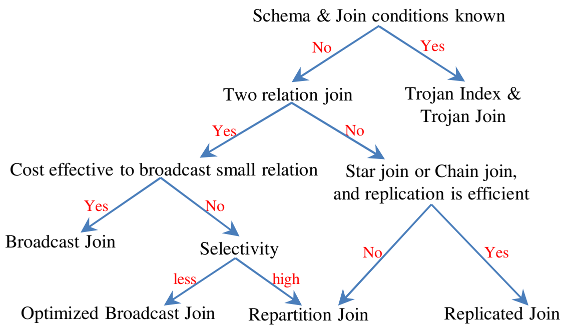

Schema & Join conditions known
No
Yes
Two relation join
Trojan Index & Trojan Join
Yes
No
Cost effective to broadcast small relation
Star join or Chain join, and replication is efficient
Yes
No
Broadcast Join
Selectivity
No
Yes
less
high
Optimized Broadcast Join
Repartition Join
Replicated Join

Fig. 5. Decision tree for Join algorithm selection

In case of Prior knowledge of schema and join condition, Trojan index and Trojan join should be used for better performance. Multiway join (Replicated join) would be efficient in case of star and chain join between more than two relations, otherwise performing cascade of two way join would be better. For join between two relations such that one relation is smaller and efficient to transmit over network then Broadcast join would be a good choice. When less number of tuples of a relation contributes to join result then prefer Optimized Broadcast Join or Semi join, else perform Repartition join.

6 Comparison

Consider that a join is performed between relations R1(a,b) and R2(b,c). Table 1 compares above mentioned join algorithms based on number of MapReduce jobs required for execution, advantages of using a particular method and issues involved.

Table 1. Comparison of Join processing methods

Join Type	MapReduce jobs	Advantages	Issues
Repartition	1 MapReduce job	Simple implementation of Reduce phase	Sorting and movement of tuples over network.
Broadcast	1 Map phase.	No sorting and movement of tuples.	Useful only if one relation is small.
Optimized Broadcast	2 jobs for Semijoin, 1 Map phase for Broadcast.	Size of large relation can be reduced and broadcasted.	Extra MapReduce jobs are required to perform semi join.
Trojan	1 Map phase.	Uses schema knowledge.	Useful, if join conditions are known.
Replicated	1 MapReduce job.	Efficient for Star join and Chain join.	For large relations more number of reducers / replicas are required.

7 Conclusion

This paper has described current research work done for optimization of the join query processing in MapReduce environment. Lots of research work is already done in Distributed databases, Parallel databases and Relational databases; which could be utilized for further improvement of join query execution in MapReduce environment. Algorithms described above do not provide generic solution which would give optimized performance in all cases. So, we proposed a strategy for join algorithm selection which could be applied dynamically based on the various parameters like size of relation; knowledge of schema and selectivity.

References

1. Jeffrey, D., Sanjay, G.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (2004)
2. Apache Foundation – Hadoop Project, `http://hadoop.apache.org`
3. Miao, J., Ye, W.: Optimization of Multi-Join Query Processing within MapReduce. In: 2010 4th International Universal Communication Symposium, IUCS (2010)
4. Foto, N.A., Jeffrey, D.U.: Optimizing Multiway Joins in a Map-Reduce Environment. IEEE Transactions on Knowledge and Data Engineering 23(9) (2011)
5. Spyros, B., Jignesh, M.P., Vuk, E., Jun, R., Eugene, J., Yuanyuan, T.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: SIGMOD 2010, June 6–11. ACM, Indian-apolis (2010)
6. Jens, D., Jorge-Arnulfo, Q., Alekh, J., Yagiz, K., Vinay, S., Jorg, S.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). In: Proceedings of the VLDB Endowment, vol. 3(1) (2010)
7. Sai, W., Feng, L., Sharad, M., Beng, C.: Query Optimization for Massively Parallel Data Processing. In: Symposium on Cloud Computing (SOCC 2011). ACM, Cascais (2011)
8. Yang, H.-C., Dasdan, A., Hsiao, R.-L., Parker, S.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In: SIGMOD 2007, June 12–14. ACM, Beijing (2007)
9. Minqi, Z., Rong, Z., Dadan, Z., Weining, Q., Aoying, Z.: Join Optimization in the MapReduce Environment for Column-wise Data Store. In: 2010 Sixth International Conference on Semantics, Knowledge and Grids. IEEE (2010)
10. Konstantin, S., Hairong, K., Sanjay, R., Robert, C.: The Hadoop Distributed File System. In: MSST 2010 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST (2010)