ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

Comparison of Join Algorithms in Map Reduce Framework

Mani Bushan[1], Balaraj J[2], Oinam Martina Devi[3]

Assistant Professor, AIMIT, St Aloysius College, Mangalore, India[1]

M.Sc.(ST) III Semester, AIMIT, St Aloysius College, Mangalore, India[2]

M.Sc.(ST) III Semester, AIMIT, St Aloysius College, Mangalore, India[3]

ABSTRACT: In the current technological world, there is generation of enormous data each and every day by different media and social networks. The MapReduce framework is increasingly being used widely to analyse large volumes of data. One of the techniques that framework is join algorithm. Join algorithms can be divided into two groups: Reduce-side join and Map-side join. The aim of our work is to compare existing join algorithms which are used by the MapReduce framework. We have compared Reducer-side merge join and Map-side replication-join in terms of pre-processing, the number of phases involved, whether it is sensitive to data skew, whether there is need for distributed Cache, memory overflow.

I INTRODUCTION

Large-scaled data warehouse systems, data-intensive analysis, cloud computing technologies have been used by Data-intensive applications. Large-scale data analysis applications use MapReduce paradigm [6] to process the data. MapReduce Framework is a programming model that is used to process and generate large data sets. User specifies a map function that processes a (key value) pair to generate a set of intermediate (key value) pairs and a reduce function which will merge all the intermediate values associated with the intermediate key [5]. Let us look upon the execution of MapReduce execution.

MapReduce Execution:

The Map/Reduce framework consists of two operations, “map” and “reduce”, which are executed on a cluster of shared-nothing commodity nodes. In a map operation, the input data available through a distributed file system, is distributed among a number of nodes in the cluster in the form of key-value pairs. Each of these mapper nodes transforms a key-value pair into a list of intermediate key-value pairs [1]. The intermediate key-value pairs are propagated to the reducer nodes such that each reduce process receives values related to one key. The values are processed and the result is written to the file system [1].

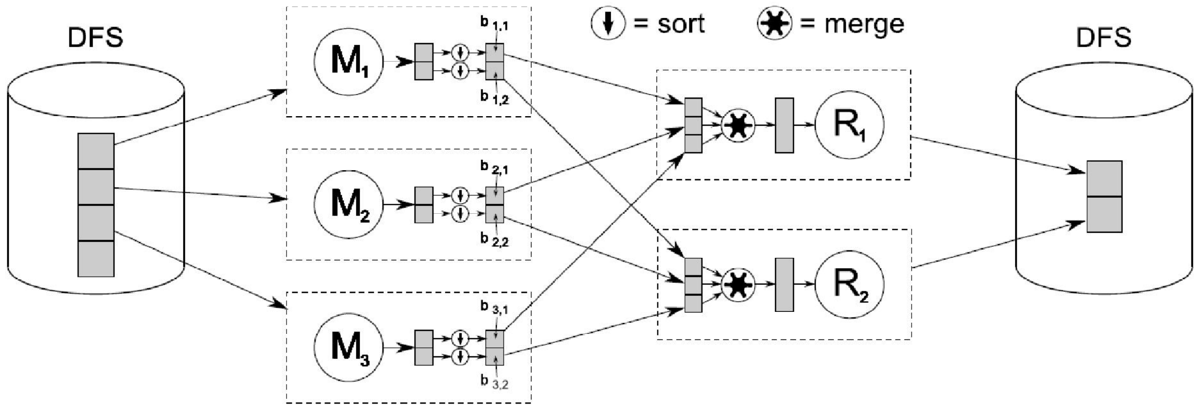


Figure 1 MR execution in detail [3].

In [3], the authors have described crucial implementation details of a number of well-known join strategies in MapReduce, and present a comprehensive experimental comparison of these join techniques on a 100-node Hadoop cluster. The authors have provided the overview of MapReduce overall. They have described how to implement several equijoin algorithms for log processing in MapReduce. They have used the MapReduce framework as it is, without any modification. Therefore, the support for fault tolerance and load balancing in MapReduce is preserved. They have worked on Repartition Join, Broadcast Join, Semi-Join, and Per-Split Semi-Join. The authors have revealed many details that make the implementation more efficient. We have evaluated the join algorithms on a 100-node system and shown the unique tradeoffs of the mentioned join algorithms in the context of MapReduce. We have also explored how our join algorithms can benefit from certain types of practical preprocessing techniques.

In [4], the authors haveexamined the algorithms to perform equi-joins between the datasets over MapReduce and have provided a comparative analysis. The results shows that all join algorithms are significantly affected by certain properties of the input datasets and that each algorithms perform better under particular circumstances. Our cost model manages to capture these factors and estimates fairly accurately the performance of each algorithm.

II COMPARISON OF ALGORITHMS

Join Algorithms:

In Join, there are two types of join algorithms:

- Map-Side Join Algorithm
- Reduce-side Join Algorithm

Map-Side join:

It is an algorithm that doesn’t make use of reduce phase. This kind are of two types: partition join and memory join. Partition join is when the data is previously partitioned with the same partitioner. The relevant parts are joined during the Map phase. This is sensitive to data skew. Memory join is when the whole data set is sent to all mappers if dataset is smaller but dataset is partitioned over the mappers if it is bigger dataset [6].

Reduce-side join:

It is an algorithm that does pre-processing of data in Map phase and direct join is done during the Reduce phase. This type of join is the most general without any restriction on the data. Reduce-side join is very much time-consuming, because it has an additional phase that transmits the data over the network from one phase to another. The algorithm has to pass the information about the source of data through the network. The Disadvantage of these algorithms are sensitivity to data skew.Thiscan be addressed by replacing the default hash partitioner with the range partitioner. In this group there are three kindsofalgorithms: General reducer-side join,Optimized reducer-side join,the Hybrid Hadoop join [6].

Comparison:

Data-intensive applications required to process multiple data sets. This implies the need to perform several join operation. Its known join operationsarethe most expensive operations in terms both I / O and CPU costs [6]. Now let us see two of the reducer side join algorithms analysed in the earlier work:

General reducer-side join:

It is the simplest join algorithm. The algorithm has Map and Reduce phases. The records from two data sets with the same key will land on the same reducer, which will then do a Cartesian product. The below is the pseudo code of the algorithm [3]:

```
Map (K: null, V : a record from a split of either R or L)
        join_key← extract the join column from V
        tagged_record← add a tag of either R or L to V
        emit (join key, tagged record)
```

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014

```
Reduce (K′: a join key, LIST_V ′: records from R and L with join key K′)
    create buffers BR and BL for R and L, respectively
    for each record t in LIST V ′ do
        append t to one of the buffers according to its tag
    for each pair of records (r, l) in BR × BL do
        emit (null, new record(r, l))
```

Working: This join can be implemented in one MapReduce job. In the map phase, each map task works on a split of either R or L. To identify from which table an input record comes, each mapper task tags the record with its originating Table, and extracted join key and the tagged record as a (key, value) pair is been output. Then the framework partitions, sorts and merges the outputs. The reducer is fed with all the records for each join key after grouping them together. For each join key, the reducer first separates and buffers the input records into two sets according to the table tag and then performs a cross-product between records in these sets [3].

The problems of this join algorithm is the reducer should contain sufficient memory for all records to hold with the same key. When the key cardinality is small or when the data is highly skewed, all the records for a given join key may not fit in memory [3]. The algorithm is also sensitivity to the data skew [6].

Map-side partition join:
The abbreviation is MSPJ**.** This algorithm assumes that the two sets of data pre-partitioned by the same partitioner. This is also known as default map join. At the Map phase one of the sets is read and loaded into the hash table, then two sets are joined by the hash table. It also buffers all records with the same keys in memory. This improvises the previous join algorithm by overriding sorting and grouping by the key as well as tagging data source. It is also called as Improved Repartition Join in [3].

The below is the pseudo code of the algorithm [3]:

```
Map (K: null, V : a record from a split of either R or L)
    Join_key←    extract the join column from V
    Tagged_record←   add a tag of either R or L to V
    Composite_key←  (join key, tag)
    emit (composite key, tagged record)


Partition (K: input key)
    Hashcode← hash func(K.join key)
    returnhashcode mod #reducers

Reduce (K′: a composite key with the join key and the tag, LIST V ′: records for K′, first from R,   then L)
    create a buffer BR for R
    for each R record r in LIST V ′ do
        store r in BR
    for each L record l in LIST V ′ do
        for each record r in BR do
        emit (null, new record(r, l))
```

Working: First, in the map function, the output key is changed to a composite of the the table tag and join key. The table tags are generated in a way which ensures the records from R will be sorted ahead of those from L on a given join key. Second, the Partitioning function is customized so that the hash code is generated from just the join key part of the composite key.

This way the same join key records are still assigned to the same reduce task. The grouping function in the reducer is customized so that the records are grouped on just the join key. Finally, as records from the smaller table R are guaranteed to be ahead of those from L for a given join key, only R records are buffered and L records are streamed to generate the join output.

This fixes the buffering problem present in the GRSJ but both methods include two major sources of overhead that can affect the performance.

Comparison:
The features of the algorithm are presented in the Table 1. The pre-processing approaches is good when the data is prepared in advance for example it comes from another MapReduce job. Algorithms with only one phase and without tagging are more preferred due to the fact that no additional transferring of data is required through the network. Approaches that sensitive to the data skew can be improved by optimizations with range partitioner. Semi-join algorithms can be used to improve the performance in case of data low selectivity and reduce the possibility of memory overflow.

	The number of phases	**Tags**	**Sensitive to data skew**	**Memory overflow**	**Join algorithm**
GRSJ	2	To value	Yes	Number tuples for the same key is large	Nested loop
ORSJ	2	To Key and Value	yes	Number tuples for the same key is big	Nested loop

Table 1: Comparative analysis of algorithms.

Figure 1: Executions time of different phases of algorithms. Size 10^4*10^5.

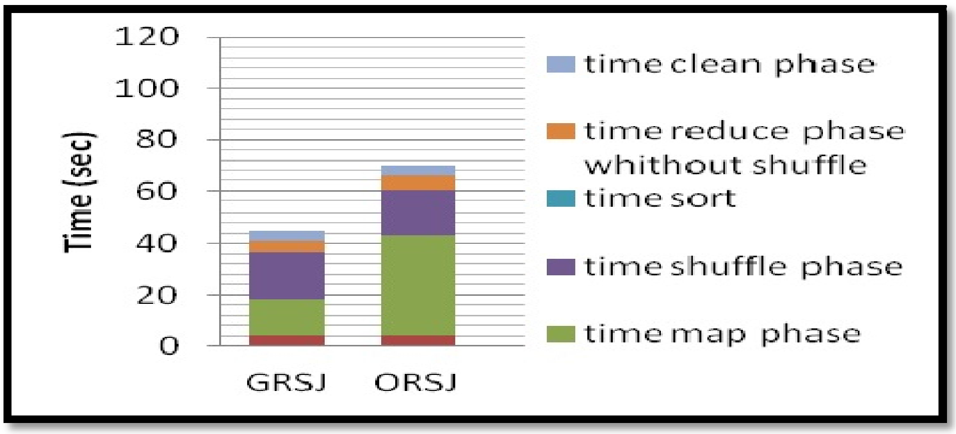


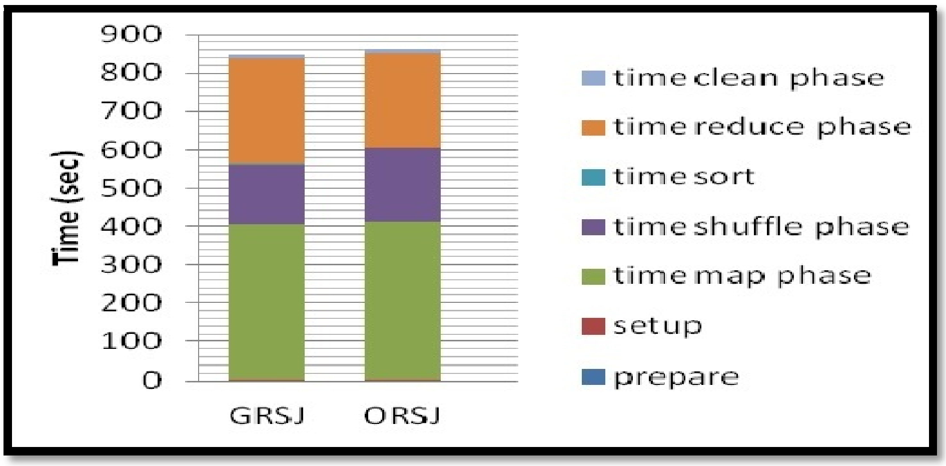
Figure 2: Executions time of different phases of algorithms. Size 106*106.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol.2, Special Issue 5, October 2014



By analysing the above results we can come to a conclusion that both the algorithm works in the same way. The drawback of the GRSJ algorithm is that, all the record for the join key has to be buffered. Thus, it needs larger memory. This drawback is overcome by ORSJ. However, the sensitive to skew data is the problem in both algorithms.

III CONCLUSION

The MapReduce framework is increasingly used in the Big Data Technology. Joining all type of reference data in the MapReduce framework has become an important part of the analytic operations for most of the enterprise customers. The performance of the various join algorithms in MapReduce differs in their own way. The above comparison has been done according the analysis by the authors of the respective papers. The future direction is that we need to implement the proposed algorithms.

REFERENCES

1. Fariha Atta. Implementation and analysis of join algorithms to handle skew for the hadoopmapreduce framework. Master's thesis, MSc B Informatics, School of Informatics, University of Edinburgh, 2010.
2. ShivnathBabu.Towards automatic optimization of mapreduce programs. Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.
3. Spyros Blanas, Jignesh M. Patel, VukErcegovac, Jun Rao, Eugene J. Shekita, and YuanyuanTian.A Comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 international conference on Management of data, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.
4. A Chatzistergiou. Designing a parallel query engine over map/reduce. Master's thesis, MSc Informatics, School of Informatics, University of Edinburgh, 2010.
5. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. Commun. ACM, 53:72–77, January 2010.
6. *Pigul*. Comparative Study Parallel Join Algorithms for MapReduceenvironment.Proceedings of the Institute for System Programming of Russian Academy of Sciences *Saint Petersburg State University.2012*
7. ShivnathBabu. Towards automatic optimization of MapReduceprograms.In SIGMOD '10: Proceedings of the 2010 international conference on Management of data. Pages 137-142. New York, NY, USA, 2010. ACM.
8. Google Research Publication: MapReduce. (n.d.). Retrieved from google: http://research.google.com/archive/mapreduce.html