Handling Skewed Data in Mapreduce Environment

A. Arun Kumar\textsuperscript{1} and K. Venkata Ramana\textsuperscript{2}

\textsuperscript{1}A. Arun Kumar, Department of CS&SE, Andhra University College of Engineering (A), Visakhapatnam, India
\textsuperscript{2}K. Venkata Ramana, Department of CS&SE, Andhra University College of Engineering (A), Visakhapatnam, India

ABSTRACT

The MapReduce framework is a parallel processing paradigm which uses shared nothing architecture for processing big data on a distributed cluster. However, MapReduce systems have become well known for processing large data sets and are more significantly being used in scientific applications. In contrast to simple application scenarios, scientific applications involve complex computations which poses challenge to MapReduce systems. Particularly because, (a) processing time complexity of the mapreduce task is generally high and (b) scientific data is frequently skewed. When the data is heavily skewed, the data affects the performance of operations in parallel environment (such as mapreduce) where data is distributed among parallel nodes for processing. Data skew can greatly limit the efficiency of parallel processing when some processing nodes are overloaded during data distribution and hence take a greater time for completion as compared to other processing nodes. This also results in wastage of resources of the idle processing nodes. As data skew naturally occurs in many applications, handling it is an important issue for improving the performance of the join operation. We implemented a skew partitioning algorithm which has the ability to handle skew and conducted rigorous experiments, to prove that our method gives efficient results when compared to the other optimization methods.

Keywords: MapReduce, Skewed Data, Reducing Skew, Fine Partitioning, Dynamic Fragmentation.

1. INTRODUCTION

Data analysis has become an important feature of businesses and organizations which can help them grow. This data analysis can produce predictions from already obtained data. MapReduce yields good results, as analysis is being performed on huge amounts of business data which is tedious. Using mapreduce the processing time decreases drastically. Mapreduce involves processing of data on multiple nodes, using key, value pairs.

Huge amount of data are collected in many areas, such as medical, finance, communication and governments. In present scenario there are urgent needs to analyze big data in these applications. However, solutions are strictly based on conventional distributed or parallel databases are very difficult to meet the needs of big data analysis. MapReduce is a prominent programming model above shared nothing architecture for processing big data with a parallel, distributed algorithm on a cluster. In present Scenario, thousands of software applications for processing big data have been implemented by this approach including large-scale image processing, machine learning as well as many other areas.

While using Big data on huge volumes of data on distributed framework, there occurs situations where some of the nodes take more time for processing while others take less. In order to attain an optimal processing time, there should be equal distribution of data among different nodes which helps in processing data efficiently at optimal time. When a certain pair of key, value pair involves higher number of computations then the processing time increases. In Order to reduce this skewed data there should be a mechanism for controlling it, using the mapper and reducer framework the load on different nodes can be equally balanced and shared.

2. RELATED WORK

The problem of skew in distributed-parallel databases has been extensively studied by many researchers in various contexts. Specifically, these references range from low-level system support (e.g., network switches or operating system
features), to optimizing data distribution among processors, to scheduling partitions in an optimal order, to improving a specific join algorithm.

DeWitt et al. proposed a practical approach to handle skew in a parallel join. They considered combinations of range partitioning (with subset-replication, weighting), load scheduling (e.g., round-robin, longest processing time (LPT)), and virtual processors (i.e., create many logical partitions and schedule multiple partitions per physical processor). For range partitioning, DeWitt et al. proposed an efficient sampling technique to estimate the distribution of the join key value. For LPT scheduling, they used a simple cost model to estimate the cost of each partition. As it is relying on range partitioning, this technique can handle redistribution skew but not the join product skew very well. The technique is easy to implement yet effective in practice.

Xu et al. proposed a parallel join approach, partial redistribution partial duplication (PRPD) to handle data skew. The PRPD approach first splits the rows of joining relation into three disjoint groups that are handled differently (redistributed, duplicated, or kept local). The rows having skewed values are kept local, the rows bearing skewed values of joining attributes in the other relation are duplicated, and the remaining rows are redistributed as in an ordinary parallel hash join. The final join is computed by union of three joins (two replicated-joins for skewed join values of each relation, and an ordinary parallel join for non-skewed values).

Gufler et al. used techniques aimed to equally distribute the partitions to the available reduce tasks. They proposed two load balancing approaches, that are based on cost model and can deal with both complex reduce tasks and skewed data. One method dynamically splits large partitions into smaller portions and replicates data if necessary and second method produces a fixed number of data partitions.

Kwon et al. presented an automatic skew mitigation approach for user defined MapReduce programs and presents SkewTune, a system that implements this approach as a drop-in replacement for an existing MapReduce implementation.

3. **SKEW IN MAPREDUCE**

The skew is differentiated from that occurs during the map-phase and skew that occurs during the reduce-phase in MapReduce.

3.1 **Skew during the Map-Phase**

There are three different reasons, why skew occurs during the map-phase. It is assumed that all the mappers process approximately the same number of input values. Therefore skew cannot occur as a result of an uneven distribution of input values.

The first type of skew is called *Expensive Record*. This sort of skew occurs, if the procession costs of individual input values, depending on the specific map-function, are significantly larger than others.

The second reason is very similar to the first reason. Since in some cases, the same application can be used to process different kinds of input data, skew can occur, because different data takes an unequal amount of time. This type of skew is called *Heterogeneous Maps*.

The third type of skew is called *Non-Homomorphic Maps*. Jobs in Hadoop can be processed successively. In some cases the data is already mapped by the prior job and map-phase of the later job is modified to process a reduce-side task. In cases, where the map phase executes a job, which is normally executed by the reduce-phase, the same skews as in the reduce-phase can occur.

3.2 **Skew during the Reduce-Phase**

During the reduce-phase two types of skews can occur. The first is similar to the *Expensive Record*. This means, that the evaluation of the reduce-function for different values is more expensive for one value than it is for another value. This problem is even more pronounced during the reduce-phase, because for a single key, a list of values has to be evaluated. This sort of skew is called *Expensive Input*.

Another sort of skew during the reduce-phase is *Partitioning Skew*. In MapReduce algorithms the outputs of the map-phase are distributed among reduce tasks using a partitioning logic. This partitioning logic can be implemented either user-defined or using a default hash-partitioning. Whereas the default hash-partitioning in most cases distributes the load of work adequate, user-defined logic can fail to achieve this goal. In both cases skew during the reduce-phase can arise in practice and can lengthen the task execution.

3.3 **Impact of parallelization**

The massive growth in the input data to be processed hampers the performance of the applications executing on uni-
processor machines. If it curtails the performance of single stream operators (selection, projection, aggregation etc.), it doubles the trouble for the join operator which handles two data streams in parallel. Matching the records from gigantic data streams is clearly overwhelming.

In large data warehousing applications, this may mean joining of trillions of records. Multi-processor or distributed processing is the solution to this problem and significantly improves the response time. In a multi-processor or distributed setting, the performance of the join operation can be improved by using partition-wise joins. The input data is partitioned among a number of machines such that processing at parallel machines can be carried out independently. For parallel evaluation of the join operator, the two datasets are partitioned by applying a hash function in the same way such that each machine handles a subset of keys which can be joined independently.

The partitioning of the datasets is performed on the basis of the join key so that a machine gets all tuples with same join key from both datasets. Thus partitioning the input datasets scatter them across a number of machines where a partition-wise join is carried out. This partition-wise join is a key to achieving scalability for massive join operations as it reduces the response time. The amount/degree of parallelism of the partition-wise join is limited by the number of partition-wise joins that can be executed concurrently. The greater the number of such concurrent partition-wise joins, the greater is the degree of parallelism. If the number of parallel partition-wise joins is 8, the degree of parallelism is 8.

Sort-merge and hash joins are the natural choices for the join operation in distributed environments since both of these join techniques can operate independently on subsets of join keys. As each partition contains the same join keys from both datasets, employing sort-merge or hash join techniques exploits the parallel partitioning and hence provides scalability and divisibility. New partitions can be added for processing without affecting the on-going processing. However, in comparison, performance of the hash join is better than that of the sort-merge join since hash joins have linear cost, as long as a minimum amount of memory is available.

### 3.4 Skewed Data and its impact

In databases, it is common that certain attribute values occur more frequently than others. This is referred to as “data skew”. Skew in the input data can limit the effectiveness of parallelization of the join query. As discussed earlier, parallelizing a join operation consists of the following steps:

1) Tuples are read from disk
2) Selection and projection are carried out on the basis of query
3) Tuples are partitioned among the parallel sites
4) Tuples of partitions on each site are joined.

Skew can occur at any of these stages and hence categorized as the tuple placement skew, selectivity skew, redistribution skew, and join product skew for each of the above stages respectively.

The initial placement of tuples in partitions may vary, giving rise to the tuple placement skew. The selectivity skew results from the fact that applying a selection predicate on different partitions may result in a varying number of selected tuples left in each partition. The redistribution skew is caused by varying number of tuples in the partitions after applying the redistribution scheme for partitioning. The join product skew is the result of differences in join selectivity at each node. For our implementation, we are not considering the tuple placement skew since MapReduce creates the file splits of almost even sizes. The selectivity skew is also ignored because firstly, it does not have any considerable impact on the performance and secondly, we assume in our programs that the selection and projection predicates are not applied.

The join product skew cannot be avoided because it is evident only after the partitions of the two relations are joined. The redistribution skew is the most important and major type of skew that impacts the load distribution among nodes. This skew is caused by the selection of an inappropriate redistribution strategy for partitioning. In further discussions of skews, we will be referring only to the redistribution skew and will handle only this skew in our implementation. After the redistribution of tuples in partitions, the hash join algorithm is applied on the partitions of the two datasets at each node. Although the hash join algorithm is easily divisible and scalable, it is very sensitive to skew. Skew in the keys results in the variance in time taken by processing nodes. If some keys appear quite frequently in the input relation, an
overly used key is sent to only one processing node on the basis of the hash partitioning. This results in an uneven distribution of the keys since the partitions receiving overused keys will contain too many tuples. As a result, the nodes processing these partitions take too much time for completion and hence become a performance bottleneck. The performance of the whole distributed system is adversely affected by these heavy-hitter nodes as some nodes remain underutilized. To take full benefit from the parallel distributed environment, it is therefore important that the redistribution strategy should be selected in such a way that partitions are of considerable size and evenly distributed to avoid the load imbalances. Current partitioning strategies are divided into two categories: hash partitioning and range partitioning. Both partitioning strategies have different sensitivities to different degrees of skew in the input keys. In the following discussion, we observe their sensitivities and conclude which partitioning strategy is most effective for skew handling and should be incorporated in our algorithm.

4. SKEW DATA REDUCTION

4.1 Skewed Data Processing

A MapReduce job is modelled as a sequence of invocations of **M map** and **R reduce** tasks (shown in Figure). Tasks are modelled as follows: 
**map**(**k**1,**v**1) ⇔ [**k**2,**v**2] and 
**reduce**(**k**2,[**v**2]) ⇔ [**k**3,**v**3]. **Map** tasks take as input (**k**1,**v**1) pairs and return a list of (**key**,**value**) pairs of possibly different types, **k**2 and **v**2. The values associated with the same key **k**2 are grouped together into a list and passed as input to the appropriate **reduce** task, which emits arbitrary (**key**,**value**) pairs of a final type, **k**3 and **v**3. All (**k**2,[**v**2]) pairs processed by the same **reduce** task on a cluster’s node, are considered a partition. The usage of a larger number of partitions compared to the number of **reduce** tasks to minimize the skewness of the intermediate data.

The number of computations made by each split is scanned and distributed equally among the different participating nodes. This achieves load balancing during the Map phase of the process.

**Partitioning Schemes**

Two types of partitioning approaches are described below.

**Fine Partitioning**

Fine Partitioning is based on the fact that more partitions than reducers have been created. This allows the system a degree of freedom to distribute them. To distribute the partitions to the reducers, the algorithms takes the biggest partition not yet assigned to a reducer and distributes this partition to the reducer with the smallest load. This step is repeated until all
partitions have been assigned to a reducer. This is a relatively simple algorithm to distribute the partitions to the reducers.

**Dynamic Fragmentation**

Because the fine partitioning strategy discussed may be ineffective in some cases, a second strategy is presented. This strategy takes partitions, which exceed the average partition size by a predefined factor, and splits them into smaller fragments. Because this fragmentation is carried out locally for each mapper, the information is transferred to a central unit. The central unit decides to fragment the partition over the whole system or ignore the fragmentation. If the central unit decides to split the partition up and reduce it by two different reducers the algorithm ensures that the necessary data and information are forwarded to the affected reducers.

### Algorithm 1

**Algorithm : 1**

```
Read (K, [V]) - Key Value Pairs
1: for (i=1; K != ∅; i++)
2:     for (i=1; K != ∅; i++)
3:         c = Number of Values in all (K, [V])
4:     Split the data into splits depending on nodes given and count
5: end
6: return count;
```

### Algorithm 2

**Algorithm : 2**

```
Read (K, [V]) - Key Value Pairs
1: if K has been assigned to node j then
2:     Assign the pair to node j
3: else
4:     Assign the pair to the node with least load
5: end if
```

Algorithm 1 is used for evaluating the number of key, value pairs participating and to count the number of pairs, in order to split them among the nodes.

Algorithm 2 is for assigning each and every key, value pair to the nodes in a load balancing approach at the nodes.

### 4.2 Explanation of the Algorithm

Acquiring information about the data distribution and adjust the work according to this information is only one way to deal with skew. The procedure of load balancing of key value pairs is achieved by considering every key and one of the values in the value set as one individual element. A counter is placed in order to compute the total number of elements are present. When the total number of elements are evaluated then this value is used for splitting the data into equal splits with respect to number of computations. The number of nodes involved in processing also used to define how to split the data. When the data is splitted each split is assigned to the different participating nodes.

Some partitions may grow excessively large, making a good load balancing impossible. Which requires dynamically splitting very large partitions into smaller *fragments*. By defining a partition to be very large if it exceeds the average partition size by a predefined factor control can be achieved. Similar to partitions, fragments also contains multiple clusters. In contrast to partitions, the number of fragments can vary from mapper to mapper.
As before, every mapper starts creating its output partitions according to the count ‘c’ of values. If a partition gains excessively more weight than the others, the mapper splits this partition into fragments. Upon completion, each mapper sends a list of partitions which it has split into fragments, along with the monitoring data, to the controller. For each partition which has been fragmented on at least one mapper, the controller considers both the exploiting fragments or to ignore them. This is achieved by calculating the partition bundles for each possible combination and then picking the best one. When the fragments of a partition are sent to different reducers, data from mappers which have not fragmented that partition needs to be replicated to all reducers which get assigned one of the fragments. A filtering step is inserted at the reducer side that eliminates data items not belonging to the fragments of that reducer immediately after receiving the file.

5. EXPERIMENTATION RESULTS

5.1 Dataset Evaluation
Data are the set of tuples, which attributes are separated by a tab space. In this experiment we use five datasets. Tuple is split into a pair of a key and a value, where value is the remaining attributes. Other dataset used as dataset is taken from sales_data database in which each attribute is separated by a comma(,) these files are with the extension of .csv(comma separated values).

<table>
<thead>
<tr>
<th>Name of Dataset</th>
<th>No.of Tuples</th>
<th>Delimiter</th>
<th>Execution Time (in mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_0</td>
<td>100000</td>
<td>\t (tab space)</td>
<td>3.5</td>
</tr>
<tr>
<td>data_1</td>
<td>99998</td>
<td>\t (tab space)</td>
<td>3.0</td>
</tr>
<tr>
<td>data_2</td>
<td>20000</td>
<td>\t (tab space)</td>
<td>1.5</td>
</tr>
<tr>
<td>data_3</td>
<td>2155</td>
<td>\t (tab space)</td>
<td>1</td>
</tr>
<tr>
<td>salesdata.csv</td>
<td>210000</td>
<td>\t (tab space)</td>
<td>5.5</td>
</tr>
</tbody>
</table>

5.2 Operating Environment
Operating environment consists of a standalone system, which is capable of running individual virtual machines. Each machine consists of namenode, secondary node and a datanode. We evaluated the implementation on a single node running Hadoop 0.21.0.

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>1.8 Ghz</td>
<td>2.3 Ghz</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Hard Disk Space</td>
<td>50 GB</td>
<td>100 GB</td>
</tr>
<tr>
<td>Display</td>
<td>800 x 600 256 colours</td>
<td>1024 x 768 High Colour (16-bit)</td>
</tr>
</tbody>
</table>

Figure 2. Execution times of datasets

5.3 Processing times At Different Nodes
Each dataset used in this thesis are derived from various transaction databases obtained from various sources. Datasets consists of transactions which lists out all the items bought in a particular transaction. Execution time for datasets varies depending on number of records in it and depending on data skews.
Reducing Skew Data can be achieved at various phases of the MapReduce framework in different levels, but selecting adequate technique depending on where it is being performed is important. The MapReduce framework facilitates parallel processing of the data distributed among processing nodes in a computing cluster. For massive datasets, parallel processing significantly reduces the response time since processing of independent subsets of work is carried out independently at distributed nodes. As every kind of processing task benefits from parallelization, so does the parallel joining of datasets. However parallel processing is vulnerable to skew formed in the datasets. If the datasets consist skewed data on some keys, some parallel nodes will take more time to accomplish the task than others and hence the whole system will wait for those overloaded nodes. Such load imbalances reduce the benefits achieved by parallelization. The roots of these load imbalances are in the partitioning stage where input datasets are distributed among the processing sites for the join operation. Selection of an inappropriate partitioning strategy results in load imbalancing. Hence, if the input data is sufficiently skewed, the overused keys are distributed among a number of processing nodes and the overall performance of the system improves.

REFERENCES
[8] Xindong Wu, Xingquan Zhu, Gong-Qing Wu and Wei Ding “Data Mining with Big Data”. In IEEE Transactions on Knowledge and Data Engineering 2014, Vol. 26, No. 1, (pp.97-107).