HANDLING DATA SKEW IN MAPREDUCE

Benjamin Gufler[1], Nikolaus Augsten[2], Angelika Reiser[1] and Alfons Kemper[1]
[1]*Technische Universität München, München, Germany*
[2]*Free University of Bozen-Bolzano, Bolzano, Italy*
{*gufler, reiser, kemper*}*@in.tum.de, augsten@inf.unibz.it*

Keywords: MapReduce, Data skew, Load balancing.

Abstract: MapReduce systems have become popular for processing large data sets and are increasingly being used in e-science applications. In contrast to simple application scenarios like word count, e-science applications involve complex computations which pose new challenges to MapReduce systems. In particular, (a) the runtime complexity of the reducer task is typically high, and (b) scientific data is often skewed. This leads to highly varying execution times for the reducers. Varying execution times result in low resource utilisation and high overall execution time since the next MapReduce cycle can only start after all reducers are done.
In this paper we address the problem of efficiently processing MapReduce jobs with complex reducer tasks over skewed data. We define a new cost model that takes into account non-linear reducer tasks and we provide an algorithm to estimate the cost in a distributed environment. We propose two load balancing approaches, fine partitioning and dynamic fragmentation, that are based on our cost model and can deal with both skewed data and complex reduce tasks. Fine partitioning produces a fixed number of data partitions, dynamic fragmentation dynamically splits large partitions into smaller portions and replicates data if necessary. Our approaches can be seamlessly integrated into existing MapReduce systems like Hadoop. We empirically evaluate our solution on both synthetic data and real data from an e-science application.

1 INTRODUCTION

Over the last years, MapReduce has become popular for processing massive data sets. Most research in this area considers simple application scenarios like log file analysis, word count, or sorting, and current systems adopt a simple hashing approach to distribute the load to the reducers.

Processing massive amounts of data is also a key challenge in e-science. However, scientific applications exhibit properties to which current MapReduce systems are not geared. First, the runtime complexity of the reducer tasks is often non-linear. Second, the distribution of scientific data is typically skewed. The high runtime complexity amplifies the skew and leads to highly varying execution times of the reducers. Thus reducers with a low load have to wait for the reducers with high load.

MapReduce jobs with high reducer complexity include data mining tasks, which are popular in e-science and often have higher polynomial or even exponential worst case complexity. Consider, for example, the Millennium simulation (Springel et al., 2005), an important astrophysical data set that contains more than 18 million trees with a total of 760 million nodes describing the evolution of the universe. Experiments with frequent subtree mining on a subset of the Millennium trees resulted in execution time differences of several hours between the reducers.

Scientific data is often skewed. Skew arises from physical properties of the observed objects (e.g., the height of patients in medical studies), from research interests focussing on subsets of the entire domain (e.g., areas with active volcanoes in geosciences), or from properties of the instruments and software employed to gather the data. In the Millennium simulation, each tree node has a mass. The mass distribution is highly skewed, with the 7 most frequent values appearing over 20 million times each, while almost 75% of the values appear no more than 10 times.

In the map phase, MapReduce systems generate (key,value) pairs from the input data. A cluster is the subset of all (key,value) pairs, or tuples, sharing the same key. Standard systems like Hadoop[1] use hashing to distribute the clusters to the reducers. Each reducer gets approximately the same number of clusters. For skewed data, this approach is not good enough since

[1]http://hadoop.apache.org

clusters may vary considerably in size. With non-linear reducers, the problem is even worse. The non-linear reduce function is evaluated for each cluster and even sets of clusters with the same overall number of tuples can have very different execution times. Processing a small number of large clusters takes much longer than processing many small clusters.

Example 1. *Assume a set of clusters consisting of four tuples. The cost of a cluster is the number of tuples to the third. If the four tuples belong to one cluster, its cost is* $4^3 = 64$. *If the set consists of two clusters with two tuples each, the cost is only* $2 \cdot 2^3 = 16$.

In this paper, we design a new cost model that takes into account non-linear reducer functions and skewed data distributions. Instead of considering only the size of the data partition (set of clusters) that is assigned to each reducer, we estimate its execution cost. This is a challenging problem since a single cluster may be produced by different mappers in a distributed manner. Computing detailed statistics for each cluster is too expensive since the number of clusters may be proportional to the data size. We estimate cluster cardinalities and their cost from aggregated statistics computed on distributed mappers.

We design two new algorithms that use our cost model to distribute the work load to reducers. The first algorithm, *fine partitioning*, splits the input data into a fixed number of partitions. The number of partitions is larger than the number of reducers, and the goal is to distribute the partitions such that the execution times for all reducers are similar. Fine partitioning does not control the cost of the partitions while they are created, but achieves balanced loads by distributing expensive partitions to different reducers. In our second approach, *dynamic fragmentation*, expensive partitions are split locally by each mapper while they are created, and tuples are replicated if necessary. As a result, the cost of the partitions is more uniform and a good load balancing is easier to achieve for highly skewed distributions.

Summarising, our contribution is the following:

- We present a new cost model that takes into account non-linear reducers and skewed data distributions, and we propose an efficient algorithm to estimate the cost in a distributed environment.
- We propose two load balancing algorithms that are based on our cost model and evenly distribute the load on the reducers. The first algorithm, fine partitioning, splits the data into a fixed number of partitions, estimates their cost, and distributes them appropriately. The second approach, dynamic fragmentation, controls the cost of the partitions while they are created.
- We empirically evaluate our techniques on synthetic data sets with controlled skew, as well as on real e-science data from the astrophysics domain.

2 DATA SKEW IN MapReduce

From a data-centric perspective, a MapReduce system works as follows. m mappers transform the input to a MapReduce job into a bag of (key,value) pairs, the *intermediate result* $I \subseteq \mathbb{K} \times \mathbb{V}$. The sub-bag of I containing all (key,value) pairs with a specific key k is a *cluster*

$$C(k) = \{(k,v) \in I\}$$

The intermediate result is split into p *partitions*. The partition for an intermediate tuple is determined by applying a partitioning function

$$\pi : \mathbb{K} \to \{1, \ldots, p\}$$

to the key of the tuple. This way, all tuples belonging to the same cluster are placed into the same partition. A partition is thus a "container", or bucket, for one or more clusters. We denote a partition j as

$$P(j) = \biguplus_{k \in \mathbb{K}:\pi(k)=j} C(k)$$

The partitions are distributed to r reducers which produce the output of the MapReduce job. All partitions assigned to the same reducer form a *partition bundle*.

A good data distribution tries to balance the clusters such that all reducers will require roughly the same time for processing. There are two aspects which need to be considered.

1. *Number of Clusters.* Some reducers might get more clusters than others, leading to larger partition bundles and longer execution times.
2. *Difficulty of Clusters.* The execution times may vary from cluster to cluster. Reducers with "difficult" clusters might take much longer to complete than reducers with "easy" clusters, even if the overall size of the partition bundles is the same.

The first of these two points can be solved by using an appropriate hash function for partitioning the data. The second point describes two challenges which can not be handled by optimal hashing: clusters of varying size and clusters of varying complexity. In the following we will elaborate on these two aspects of load balancing in MapReduce systems.

3 COST MODEL

The reducer workload should be evenly distributed to all participating nodes. This maximises resource utilisation, as no reducers remain idle, waiting for some overloaded reducers to complete. Moreover, well-balanced execution times minimise the time until job completion, because parallel processing is better exploited. Finally, similar execution times are a common (and often implicit) assumption in both scheduling and failure detection strategies proposed for MapReduce (Dean and Ghemawat, 2008; Zaharia et al., 2008).

3.1 Current Situation

In state of the art MapReduce systems, like Hadoop, every mapper partitions the share of intermediate results it creates into r partitions (i. e., $p = r$ in the partitioning function π defined in Section 2, and all partition bundles consist of a single partition only). As all mappers use the same partitioning function, tuples belonging to the same cluster are all placed into the same partition. This is visualised in Figure 1, where we have two reducers. Thus, two partitions are created per mapper. The partitions of the first mapper are shown in more detail on the left: the first partition contains four clusters, the second one holds three.

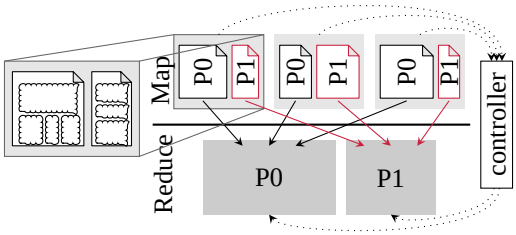


Figure 1: Traditional Data Distribution.

Typically, a hash function is used for partitioning. Assuming a reasonably good hash function, the clusters are uniformly distributed to the partitions. Every partition is then assigned to a dedicated reducer for further processing. Figure 1 shows an example of the current data distribution strategy in MapReduce: partition P0 of every mapper is assigned to the first reducer, P1 to the second one.

This approach is perfectly suitable in situations where the key frequencies are (almost) uniformly distributed, and the amount of work a reducer spends per cluster does not vary strongly. In many other situations, however, distributing the keys uniformly is suboptimal. The most prominent problems are:

1. **Skewed Key Frequencies.** If some keys appear more frequently in the intermediate data tuples, the number of tuples per cluster will vary. Even if every reducer receives the same number of clusters, the overall number of tuples per reducer will be different.
2. **Skewed Tuple Sizes.** In applications which hold complex objects within the tuples, unbalanced cluster sizes can arise from skewed tuple sizes.
3. **Skewed Execution Times.** If the execution time of the reducer is worse than linear, processing a single, large cluster may take much longer than processing a higher number of small clusters. Even if the overall number of tuples per reducer is the same, the execution times of the reducers may differ.

Example 2. *Consider a reducer which compares all items within a cluster to each other. Obviously, the reducer's complexity is quadratic in the number of tuples within a cluster. Processing a cluster with six tuples thus has a cost of* $6^2 = 36$. *Three clusters of size two only have a total cost of* $3 \cdot 2^2 = 12$. *The total number of tuples is, however, six in both cases.*

Skew is symbolised by smaller and larger partition icons and reducer boxes in Figure 1. In this example, partition P0 is much larger than partition P1 on two mappers. The reducer on the left thus gets a much larger share of the data than the one on the right.

We will now propose an approach for obtaining the best possible load balancing.

3.2 Optimal Solution

In order to balance the workload on the reducers, we need to know the amount of work required for every cluster. Typically, the work per cluster depends either on the number of tuples in the cluster, or on the byte size of the cluster, or both these parameters. Therefore, while creating the clusters, we monitor for every cluster $C(k)$ the number of tuples it contains, $|C(k)|$, and its (byte) size, $\|C(k)\|$. Based on the complexity of the reducer algorithm, we can then calculate the weight, $w(|C(k)|, \|C(k)\|)$, i.e., the amount of work for each cluster k as a function of tuple count and size.

Example 3. *For the reducer described in Example 2, we estimate the weight of a cluster as* $w(t,s) = t^2$.
As an example of a reducer complexity depending on the byte size of the processed cluster, consider the following scenario: Every tuple contains an array of values. The reducer's task is to find the median of these values per cluster over all contained tuples. The amount of work to spend thus depends on the combined size of all arrays within a cluster, rather than

the tuple count. If the array size is proportional to the tuple sizes, we will therefore base the work estimation on the byte size of the clusters. As finding the median element of an array of length n is possible in $n \log n$ time, we estimate the work as $w(t,s) = s \log s$.

We obtain the optimal, i. e., the best weight balanced assignment of clusters to reducers by solving the associated bin packing problem. The optimal solution is not feasible for two reasons.

1. In a worst-case scenario, the monitored data grows linearly in the size of the intermediate data I. Such a situation arises, e. g., when joining two tables on their primary key columns: every key value can appear only once per table, and the resulting clusters contain at most two tuples.
2. The bin packing problem is *NP* hard. Hence, even for a moderate number of clusters, calculating the assignment of clusters to reducers can become more expensive than the actual execution of the reducers.

In the following we will address these two problems, and develop heuristics for approximately solving the load balancing problem.

3.3 Approximate Cost Estimation

The first problem with the optimal solution is the size of the monitored data. In the worst case, the number of clusters, $|\mathbb{K}|$, grows linearly with the number of intermediate data tuples. With MapReduce being a system designed for processing terabyte scale data sets, we can therefore not afford to monitor every cluster individually. Instead, we do the monitoring on partition level, i. e., we create a histogram of monitoring data using the partitions $P(j)$ as histogram buckets. Besides tuple count, $t(j)$, and total size, $s(j)$, we also include the number of clusters per partition, $c(j)$ in our monitoring data μ: $\mu(j) = (c(j), t(j), s(j))$ with

$$\begin{aligned} c(j) &= |\{C(k) : k \in \mathbb{K}, C(k) \subset P(j)\}| \\ t(j) &= |P(j)| \\ s(j) &= \sum_{k \in \mathbb{K}, C(k) \subset P(j)} \|C(k)\| \end{aligned}$$

Recall from the preceding section that we need the weight for each cluster. We estimate the tuple counts and sizes of the clusters based on the monitoring information for the partitions using average values:

$$\overline{t_c}(j) = \frac{t(j)}{c(j)} \qquad \overline{s_c}(j) = \frac{s(j)}{c(j)}$$

We can now determine the processing cost per cluster, $w(\overline{t_c}(j), \overline{s_c}(j))$, using the tuple count and size estimates. Summing up all processing costs for a partition, we obtain the partition cost, $W(j)$:

$$W(j) = c(j) w(\overline{t_c}(j), \overline{s_c}(j))$$

Since the input data is assumed to be skewed, the average cost values for the clusters can differ substantially from the actual values. Despite this approximation error we achieve much better load balancing than current MapReduce implementations. We will discuss this issue in Section 4.1 and present supporting experimental evaluation in Section 6.3.

Collecting accurate statistics for clusters is an open research problem. As discussed in Section 3.2, exact monitoring at the cluster level is not feasible. Possible solutions could collect monitoring data on a granularity level between clusters and partitions, or selectively monitor only the most relevant clusters within each partition.

3.4 Distributed Monitoring

The bag I holding all intermediate tuples is not materialised on a single host. Therefore, we need to collect our monitoring data in a distributed manner, and then aggregate it in order to obtain information on the global data distribution. We denote by I_i the bag of intermediate (key,value) pairs generated by mapper i, i. e., $\{I_1, I_2, \ldots, I_m\}$ is a partitioning of I. Every mapper i gathers monitoring information for all partitions j based on the tuples in its share of the intermediate data, I_i. We collect all this monitoring data on a central controller and aggregate it in order to obtain an approximation of μ. Note that for our cost estimations we do not need to introduce a new centralised component in MapReduce, but we exploit the centralised controller for task scheduling. For the tuple count and size of partition j, we collect, on every mapper i, the local tuple count, $t_i(j)$, and size, $s_i(j)$. By summing up those values, we can reconstruct the exact number of tuples per partition, $t(j)$, and their total size, $s(j)$.

$$t(j) = \sum_{1 \le i \le m} t_i(j) \qquad s(j) = \sum_{1 \le i \le m} s_i(j)$$

For the number of clusters per partition, the same approach is not applicable, as clusters are typically distributed over multiple mappers. We employ the linear counting approach (Whang et al., 1990) for approximating the cluster count per partition.

4 LOAD BALANCING

Now that we have a cost model that takes into account non-linear reducer tasks, we define two load balancing approaches based on this model.

4.1 Fine Partitioning

By creating more partitions than there are reducers (i. e., by choosing $p > r$, in contrast to current MapReduce systems where $p = r$), we retain some degree of freedom for balancing the load on the reducers. The range of p is obviously bounded by the number of reducers, r, on the lower end, and the number of clusters, $|\mathbb{K}|$, on the upper end. With $p < r$, some reducers would not obtain any input. With $p > |\mathbb{K}|$, some partitions will remain empty.

The number of partitions, p, influences the quality of the obtained load balancing. The higher we choose p, the more possibilities the controller has to balance the load. On the other hand, the management overhead grows with p. This overhead impacts on the execution of the MapReduce job twice. First, we need to collect and process more monitoring data. For very high values of p (close to $|\mathbb{K}|$), handling the monitoring data could thus become a bottleneck in the job execution. Second, partitions are the units of data transfer (i. e., files) from the mappers to the reducers. Transferring a few large files is faster and results in less overhead than transferring many small files. We need to be aware of this trade-off when choosing p.

The goal of assigning partitions to reducers is to balance the load. The optimal load balance is achieved by solving the respective bin packing problem. Unfortunately, bin packing is *NP* hard. We propose a greedy heuristics (sketched in Algorithm 1) to determine the partition bundles. We pick the most expensive partition not yet assigned to a reducer, and assign it to the reducer which has smallest total load. The load of a reducer is the sum of the costs of all partitions assigned to that reducer. We repeat these steps until all partitions have been assigned.

Algorithm 1: Assign Partitions to Reducers.

Input: $W : \{1,\ldots,p\} \to \mathbb{R}^+$
Output: R: a set of partition bundles

1: $R \leftarrow \emptyset$
2: $P = \{1,\ldots,p\}$
3: **while** $P \neq \emptyset$ **do**
4: $\quad q = \arg\max_{j \in P} W(j)$
5: $\quad P \leftarrow P \setminus \{q\}$
6: $\quad$ **if** $|R| < r$ **then**
7: $\quad\quad R \leftarrow R \cup \{\{q\}\}$
8: $\quad$ **else**
9: $\quad\quad s = \arg\min_{l \in R} \sum_{j \in l} W(j)$
10: $\quad\quad R \leftarrow (R \setminus \{s\}) \cup \{s \cup \{q\}\}$
11: $\quad$ **end if**
12: **end while**
13: **return** R

Note that we calculate the partition bundles only after all mappers have completed their execution, which prevents the *reducer slow-start* optimisation of Hadoop. We will discuss this aspect in Section 4.3.

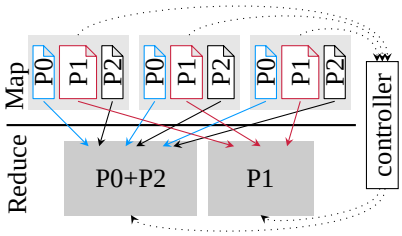


Figure 2: Partitioned Data Distribution.

An example for an approximate bin packing solution is shown in Figure 2. Even though we only have two reducers, every mapper creates three partitions. Based on the monitoring data obtained from the mappers, the controller determines the assignment of partitions to reducers. P1 is the most expensive partition and is assigned to a dedicated reducer, while P0 and P2, which are cheaper, share a reducer.

Recall from Section 3 that we do not know the exact cost for every partition, but only approximated values. This impacts the load balancing on the reducers as follows.

1. If the clusters are similar in cost, our cost estimation is accurate, and the load balanced well.
2. If the clusters are heavily skewed, i. e., there are very few clusters which are considerably larger than the others, also the partitions containing these clusters will be much larger than the others. The estimated cost for those partitions will, therefore, also be higher than that of partitions containing only small clusters. Partitions containing large clusters will thus very likely be assigned to dedicated reducers, as long as the total number of reducers is sufficiently large.
3. Finally, for moderately skewed data, two situations may arise.
   (a) The larger clusters are evenly distributed over all partitions. Then we overestimate the cost of all partition. This is, however, not a problem since the absolute cost is irrelevant for assigning the partitions to reducers and we still obtain a reasonable good load balancing.
   (b) The partitioning function assigns the larger clusters to a small number of partitions. Then the same reasoning as for heavily skewed data applies.

4.2 Dynamic Fragmentation

With the fine partitioning approach presented above, some partitions may grow excessively large, making a good load balancing impossible. In this section we present a strategy which dynamically splits very large partitions into smaller *fragments*. We define a partition to be very large if it exceeds the average partition size by a predefined factor. Similar to partitions, fragments are containers for multiple clusters. In contrast to partitions, however, the number of fragments may vary from mapper to mapper.

As before, every mapper starts creating its output partitions according to the partitioning function π. If a partition gains excessively more weight than the others, the mapper splits this partition into fragments. We choose the number of fragments, f, to be the smallest integer greater than 1 s.t. $p \not\equiv 0 \mod f$. This is shown in Figure 3. The leftmost mapper splits partition P2, which has almost twice the weight of the other partitions, into two fragments ($3 \equiv 1 \mod 2$). The mapper in the middle splits partitions P0 and P2, while on the rightmost mapper splits partition P1.

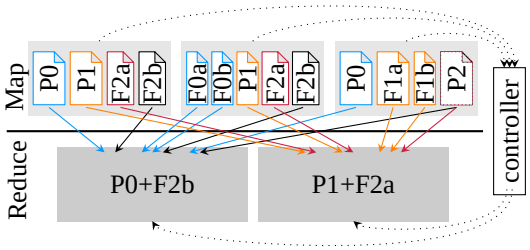


Figure 3: Fragmented Data Distribution.

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 exploiting the fragments or ignoring them. This is achieved by calculating the partition bundles (the set R in Algorithm 1) 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. In Figure 3, fragment F2a is assigned to the reducer on the right, whereas fragment F2b is assigned to the left one. Partition P2 from the rightmost mapper must be copied to both reducers, as it might contain data belonging to both 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.

We choose the best partition assignment using a cost based strategy. The first aspect that the cost function needs to consider is how well the weight is balanced. We use the standard deviation σ of the weight of the partition bundles to express this aspect. The lower the standard deviation, the better the data is balanced. The second aspect to include is the amount of replication. In the cost function we use the average weight $\overline{w}$ of the partition bundles. We want to keep $\overline{w}$, and thus the amount of replicated data, as low as possible and define the cost of an assignment R as

$$C(R) = \overline{w}(R) \cdot (1 + \sigma(R))^e$$

We strive for an assignment with low cost. The parameter e controls the influence of the balancing over replication. Low values of e favour assignments with lower replication at the cost of unbalanced partition bundles, high values favour well balanced partition bundles at the cost of replication. A good choice for e depends on the complexity of the reducer task.

Example 4. *For the reducer in Example 2 with quadratic runtime complexity, we choose a smaller value for e than for a reducer with exponential worst case complexity. The difference in execution time due to unbalanced loads is much higher for expensive reducers and the additional communication cost for replication is likely to outweighed with balanced reducers.*

In the example of Figure 3, the benefit of assigning fragments F2a and F2b to different reducers outweighed the increased cost resulting from the replication of partition P2 to both reducers. Partition P1, on the other hand, was only fragmented on the rightmost mapper. Placing its fragments on different reducers would require to replicate the partitions P1 from the other mappers to both reducers, which in our example did not pay off.

4.3 Reducer Slow-start

In "traditional" MapReduce systems, the first reducers are already launched when a small percentage of mappers is done. During this *slow-start phase*, reducers fetch their inputs from completed mappers. Both approaches presented in the preceding sections, however, require all mappers to have completed processing before the assignment of partitions (and possibly fragments) to reducers can be calculated. For highly complex reduce algorithms, the time savings due to slow-start are negligible. For reduce algorithms of moderate complexity, we can derive an initial assignment of partitions to reducers based on the monitoring data from the first completed mappers, and adapt the assignment later if necessary. Empirical evaluations

(which are not further discussed due to space limitations) show a fast convergence of the assignments after r mappers are completed.

5 HANDLING LARGE CLUSTERS

The techniques presented so far aim at distributing clusters to reducers such that the resulting load on all reducers is balanced. In some situations, however, good load balancing is not possible. Such situations arise, e. g., when we have less clusters than reducers ($|\mathbb{K}| < r$), or when the cluster costs are heavily skewed and very few of the clusters make up for most of the total cost.

According to the MapReduce processing model, a single cluster must not be distributed to multiple reducers for processing. The processing code executed on the reducers is supplied by the user. The possibilities for the framework to react to expensive clusters are therefore very limited. We propose to provide an optional extension to the interface, allowing the framework to notify the user code if expensive clusters are encountered. Hence, the user can react to large clusters with application specific solutions, e. g. using multi-threading or approximate algorithms.

6 EXPERIMENTAL EVALUATION

In this section we report on the experimental evaluation of the presented partitioning strategies and their impact on a specific e-science application.

6.1 Measurement Environment

We evaluate our partitioning strategies and their impact using both synthetic and real e-science data. We generate synthetic data sets based on Zipf distributions with 200 clusters and varying z parameter. The e-science data set consist of all *merger tree nodes* from the Millennium run[2] (Springel et al., 2005), a 294 GB data set with more than 760 million tuples.

We simulated a MapReduce environment, building on each mapper the histogram as described in Section 3.4. We then calculated the partition bundles based on these histograms, using the bin packing heuristic of Section 4.1. For the Millennium data set, we used the number of mappers (389) and the actual distribution of data to the mappers chosen by Hadoop, configured with default settings except for the HDFS block size, which we increased from 64 MB to 512 MB. We altered the block size because Hadoop chooses the number of mappers based on the number of input data blocks. With simple map tasks, it is thus reasonable to use large block sizes in order to avoid creating a huge number of short running mappers. For the synthetic data, we chose parameters close to the values observed in real world datasets. We scheduled 400 mappers, generating 1.3 million intermediate tuples each. We repeated all measurements 20 times and report the averages.

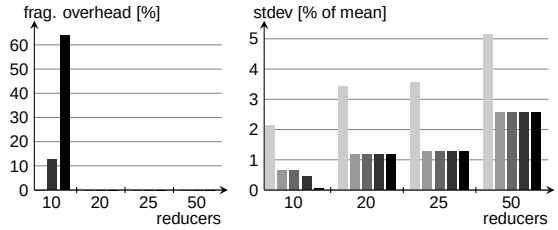
[2]http://www.g-vo.org/Millennium

6.2 Partitioning Strategies

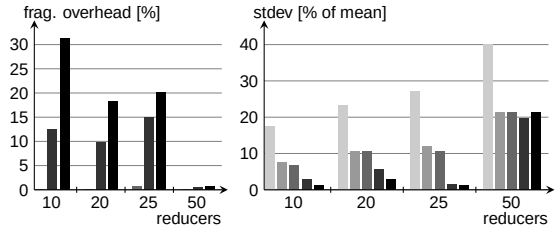
In our first evaluation, we compare the current data redistribution scheme (Section 3.1) with the fine partitioning (Section 4.1) and the dynamic fragmentation (Section 4.2) approaches for varying parameters e (0.05, 0.15, 0.3) in the cost function. We choose the number of partitions, p, to be four times the number of reducers. With this choice, we obtain a sufficient number of partitions to balance the load quite well, while not exceeding the number of clusters.

We show the obtained results for varying numbers of reducers in Figure 4. The left graph in each figure shows the replication overhead introduced by dynamic fragmentation with varying e parameter. Standard MapReduce and fine partitioning are not shown in these graphs as they introduce no replication. If dynamic fragmentation chooses the same result as fine partitioning, i. e., the fragments are not exploited, then no fragmentation overhead incurs and no bars are visible in the diagram. The right part of the figures shows the standard deviation in the tuple count per reducer. The values are relative to the average number of tuples per reducer without replication.

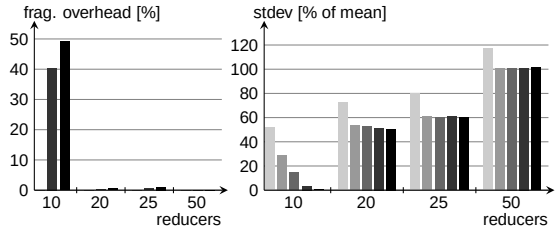
Both fine partitioning and dynamic fragmentation balance the load considerably better than standard MapReduce systems. Dynamic fragmentation has the highest impact in the scenario with moderate skew (Figure 4b) and with moderate reducer count. The remaining situations are as follows. For low skew (Figure 4a), except for the scenario with 10 reducers, no partition grows noticeably larger than the others. Therefore no fragments are created, and dynamic fragmentation falls back to fine partitioning. For high skew (Figure 4c), the partition(s) with very expensive clusters are fragmented. Expensive clusters, however, cannot be split. Therefore, the possible gain in balancedness is low, and fragments are exploited only for high e values. An exception is the scenario with 10 reducers. Due to the very low number of reducers, splitting the partition with the most expensive cluster has a strong impact, allowing to accept even high fragmentation overhead.



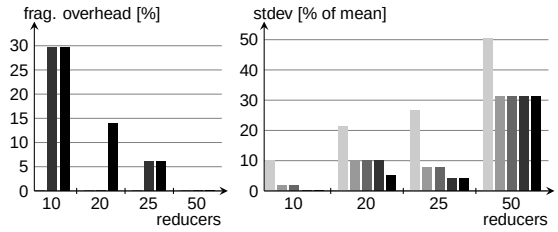
(a) Synthetic, $z = 0.1$.



(b) Synthetic, $z = 0.5$.



(c) Synthetic, $z = 0.9$.



(d) Millennium data.

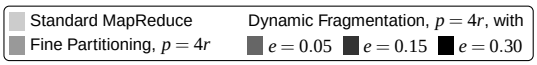


Figure 4: Data Balancing.

Comparing the different configurations for dynamic fragmentation, we see that e is a reasonable parameter for configuring the amount of replication tolerated in order to achieve better data balancing. The larger e is, the more replication is accepted if this results in better balancing. The choice of e should thus depend on the expected execution time of the reducers. For fast reducers, slightly skewed execution times are typically acceptable. For long-running reducers, more replication overhead will be outweighed by better balancing the reducer execution times.

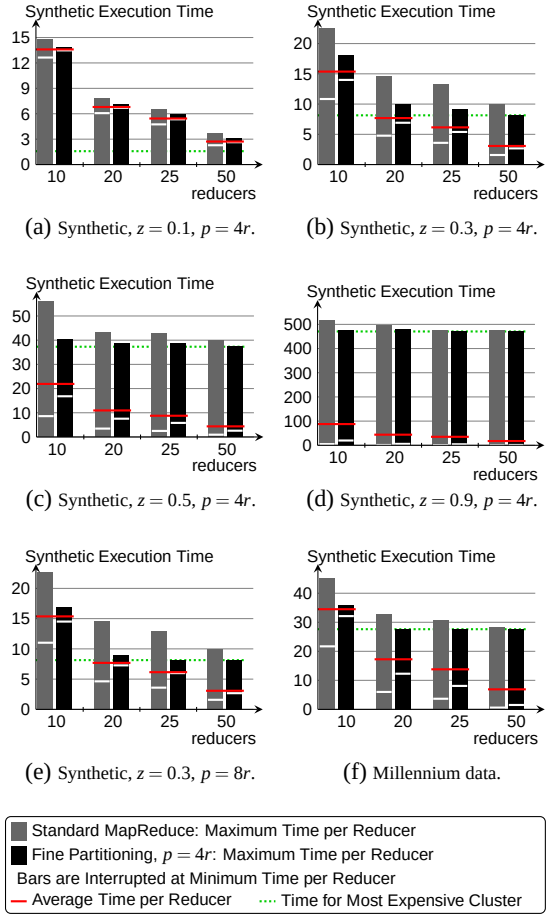
With the Millennium data (Figure 4d), the benefit of our load balancing techniques becomes even more evident. For most of the reducer numbers, even the fine partitioning approach is able to reduce the deviation by far more than 50%.

6.3 Influence on Applications

Finally, we evaluate the impact of our load balancing approaches on the execution times of a MapReduce application. Figure 5 shows the execution times for a reducer side algorithm with quadratic complexity in the number of input tuples, e. g., an algorithm doing a pairwise comparison of all tuples within a cluster. The total bar heights show the synthetic execution time for the longest running reducer. We calculate the synthetic execution time according to the algorithm complexity, based on exact cluster sizes. Bar heights up to the cut show the shortest reducer time. The (red) line spanning over both bars for the same reducer count gives the average execution time for a reducer. The average is identical for both evaluated approaches, as the same work is performed in both scenarios. Finally, every diagram includes the processing time required for the most expensive cluster (green, dotted line). This value is a lower bound of the execution time since clusters can not be split.

For the synthetic data sets, we observe the highest impact of fine partitioning on moderately skewed data ($z = 0.3$, Figures 5b and 5e). Here, we are able to reduce the time required on the longest-running reducer (the bar heights) by over 30%. For 25 and more reducers, the time spent for the longest-running reducer is very close to the most expensive cluster (green, dotted line), which gives the lowest reachable value. Note also the small difference between average (red line) and shortest (cut in the bars) execution times, indicating that only very few reducers require more time. Comparing Figures 5b and 5e, we see the positive impact of a higher number of partitions, especially for the configurations with 20 and 25 reducers.

For both very balanced data (Figure 5a) and very skewed data (Figure 5d), we see only small differences between the two approaches. In the balanced scenario, the naïve data distribution in current Hadoop obtains a reasonably good result. The average execution time (red line) is roughly half way between the shortest and the longest execution times. For the very skewed scenario, a single expensive cluster dominates the overall execution time (see the execution time for the most expensive cluster). The only way to reduce the total execution time is thus to isolate this expensive cluster. Fine partitioning achieve this goal already for a lower number of reducers.



(a) Synthetic, $z = 0.1$, $p = 4r$. (b) Synthetic, $z = 0.3$, $p = 4r$.

(c) Synthetic, $z = 0.5$, $p = 4r$. (d) Synthetic, $z = 0.9$, $p = 4r$.

(e) Synthetic, $z = 0.3$, $p = 8r$. (f) Millennium data.

Figure 5: Execution Times.

For the Millennium data set (Figure 5f) fine partitioning reaches to optimum (the maximum cost is the cost of the most expensive cluster) already with 20 reducers, while the standard approach requires more than 50 reducers.

7 RELATED WORK

Despite the popularity of MapReduce systems, which have been at the centre of distributed systems research over the last years, skew handling has received little attention. Only very recently, the SkewReduce system (Kwon et al., 2010) was proposed. In SkewReduce, data characteristics are collected by a sampling approach. The user has to provide cost functions which derive, from this sampling data, information on the runtime behaviour of the application. With the techniques described in this paper, the user needs to specify only the runtime complexity of the reducer side algorithm; all remaining components are provided by the framework.

When processing joins on MapReduce systems, data skew might arise as well. A recent publication (Afrati and Ullman, 2010) shows how to best use Symmetric Fragment-Replicate Joins (Stamos and Young, 1993) on MapReduce systems in order to minimise communication. Based on the input relation sizes, the presented system finds the optimal degree of replication for all relations. Our work is orthogonal to this approach. Skewed join attribute distribution can lead to load imbalance on the reducers, which is tackled by the techniques presented in this paper.

A scheduling algorithm for MapReduce in heterogeneous environments was presented in (Zaharia et al., 2008). They show that an improved scheduling strategy can effectively decrease the response time of Hadoop. The scheduling strategy determines invocation time and hosts for the single reduce tasks, but not the assignment of clusters to reducers. Their approach can thus be combined with our load balancing techniques in order to further reduce the response time.

MapReduce and (distributed) database systems are often used for similar tasks. Hence, over the last years there has been substantial effort from the database community to both compare (Pavlo et al., 2009; Stonebraker et al., 2010), and to combine the two approaches. Database systems are used as intelligent storage back-ends for MapReduce (Abouzeid et al., 2009), and indexing is integrated with Hadoops native data storage (Dittrich et al., 2010). MapReduce applications written in dedicated high level languages are optimised using techniques from query optimisation (Gates et al., 2009; Battré et al., 2010). All of this work is orthogonal to the data skew handling techniques for MapReduce we presented in this paper.

Distributed database literature offers much prior work on handling skewed data distributions. Our dynamic fragmentation approach is inspired by distributed hash join processing techniques (Zeller and Gray, 1990), and extends these techniques such that multiple mappers can contribute as data sources.

Data skew was also tackled in the Gamma project (DeWitt et al., 1992). Some of their techniques are applicable to MapReduce. Our fine partitioning approach is similar to *Virtual Processor Partitioning*. Other techniques are very specific to distributed join processing and cannot be directly transferred to our scenario. An example is the *Subset-Replicate* approach. Similar to the Fragment-Replicate Join, this approach allows to distribute one cluster over multiple sites. Such a technique is not applicable to arbitrary distributed grouping/aggregation tasks, which we need for load balancing in MapReduce.

Our bin packing heuristics for distributing the load

to reducers resembles the First Fit Decreasing (FFD) algorithm (Johnson, 1973). Different from the standard bin packing scenario, the bins in our scenario have no capacity limit. We choose the bin with the lowest load to place the next item in.

8 SUMMARY AND ONGOING WORK

Motivated by skewed reducer execution times in e-science workflows, we analysed the behaviour of MapReduce systems with skewed data distributions and complex reducer side algorithms. We presented two approaches, fine partitioning and dynamic fragmentation, allowing for improved load balancing.

In future work, we will consider collecting more sophisticated statistics on the partitions in order to estimate the workload per partition more accurately. Moreover, we will focus on skewed data distributions on the mappers. Such skew can arise, e. g., in data warehouses capturing a shifting trend.

ACKNOWLEDGEMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF, contract 05A08VHA) in the context of the GAVO-III project and by the Autonomous Province of Bolzano - South Tyrol, Italy, Promotion of Educational Policies, University and Research Department.

REFERENCES

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. (2009). HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*.

Afrati, F. N. and Ullman, J. D. (2010). Optimizing Joins in a Map-Reduce Environment. In *EDBT*.

Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., and Warneke, D. (2010). Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1).

DeWitt, D., Naughton, J. F., Schneider, D. A., and Seshadri, S. (1992). Practical Skew Handling in Parallel Joins. In *VLDB*.

Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. (2010). Hadoop++: Making a Yellow Elephant Run Like a Cheetah. In *VLDB*.

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U. (2009). Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. In *VLDB*.

Johnson, D. S. (1973). Approximation Algorithms for Combinatorial Problems. In *STOC*.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. A. (2010). Skew-resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SoCC*.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., and Stonebraker, M. (2009). A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*.

Springel, V., White, S., Jenkins, A., Frenk, C., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., and Pearce, F. (2005). Simulating the Joint Evolution of Quasars, Galaxies and their Large-Scale Distribution. *Nature*, 435.

Stamos, J. W. and Young, H. C. (1993). A Symmetric Fragment and Replicate Algorithm for Distributed Joins. *IEEE TPDS*, 4(12).

Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., and Rasin, A. (2010). MapReduce and Parallel DBMSs: Friends or Foes? *CACM*, 53(1).

Whang, K.-Y., Zanden, B. T. V., and Taylor, H. M. (1990). A Linear-Time Probabilistic Counting Algorithm for Database Applications. *TODS*, 15(2).

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. (2008). Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*.

Zeller, H. and Gray, J. (1990). An Adaptive Hash Join Algorithm for Multiuser Environments. In *VLDB*.