Adaptive Load Balancing in MapReduce using Flubber

Rohit Paravastu, Rozemary Scarlat, Balakrishnan Chandrasekaran
{rohit, rozemary, balac}@cs.duke.edu
Department of Computer Science
Duke University

Abstract—MapReduce has emerged as a successful framework for addressing the heavy demand for large-scale analytical data processing, in this peta-byte age. However, while on one hand the sheer size of data makes problems more challenging, the flexibility offered by the MapReduce frameworks on the other hand, makes the learning curve far steeper than expected. The general idea behind a MapReduce framework is to split the task into two components – a Mapper and a Reducer. The mapper executes a user-defined computation on chunks of data and generates the results, while the reducer groups the results together based on a common attribute. Scalability, hence, appears as an inherent trait of the design. A critical parameter in this configuration is the number of reducers required for a given task, and frameworks like Hadoop expect the user to specify this parameter while submitting a job. In this report, we focus on Hadoop and argue that deciding the number of reducers is a non-trivial task, let alone deciding it prior to running the job. To address this issue, we present Flubber – a simple pre-job that can be sandwiched between the original job and Hadoop. With a couple of parameters from the user, it takes a stab at figuring out the ideal number of reducers for the given job.

I. INTRODUCTION

In the last few years, the importance of analytical processing techniques has grown exponentially. From online recommendation systems to scientific research, we are depending on analytical processing systems to go through mountains of data to give us useful and interesting information. To scalably process these huge volumes of data is a challenging task. Cloud computing and MapReduce techniques help us in resolving this problem. These frameworks automatically parallelize the processing across a group of machines to effectively scale the running of a huge job. Recent research in databases and parallel computing has focussed on ways to make the MapReduce framework more effective in processing huge volumes of data. One such method to increase the effectiveness of MapReduce is Adaptive Load Balancing. Load Balancing is a technique where the amount of work to be done is equally divided between all the units (mappers or reducers). In a MapReduce environment, Load Balancing over the mappers is done by dividing the input data into equal chunks. But, when balancing the load among the reducer units, the amount of work to be done by the reducers cannot be estimated at the start of the job. Typically, the number of reducers and the partitioner is decided at the start of the job. This may prove to be a serious bottleneck for the running time of the job. So, there is a need to come up with techniques to adaptively balance the load on the reducers while the job is being executed. This problem is defined as Adaptive Load Balancing. The main focus of this project would be to tackle the adaptive load balancing problem by finding ways to come up with good partitions and other such techniques.

II. BACKGROUND

In a MapReduce framework, the number of reducers required for a given job configuration depends on the size of the data distribution of the mappers’ output. However, in practice, there is often no clear information on the mappers’ output distribution. This leads to the user often guessing, sometimes even blindly, the number of the reducers required. The efficiency of the job in terms of completion times, nevertheless, depends amongst other parameters, the number of reducers chosen during job submission. This, hence, is similar to the chicken or the egg conundrum. In some cases, the situation can be improved by relying on knowledge of the workload or prior experience. There are also some widely used heuristics, viz., $(0.95 \text{ or } 1.75) \times \text{number-of-reduce-slots}$, to improve the blind guesses. It is important to note that these heuristics are often agnostic to the mapper output patterns. Hence, such heuristics often fall short of delivering optimal job completion times.

Workloads can vary widely between jobs, and even with an optimal number of reducers we cannot guarantee uniform utilization of the nodes in a MapReduce cluster. In the Hadoop implementation, this load balancing issue regarding multiple jobs running is addressed by the Fair Scheduler. This plug-gable component represents a method of assigning resources such that the computation resources in a cluster is fairly shared between jobs. There also exist systems that continuously (re)balance the data generated on Hadoop uniformly across the nodes in a cluster. However, according to our knowledge, no systems currently exist that help in configuring a job optimally to get the best completion times while not exploiting or relying on any detail below the abstraction provided by Hadoop. Flubber is the result of an exercise to understand the capability of building solutions on top of Hadoop to help the user to configure jobs optimally. While Flubber currently helps in determining just one of the various job configuration

1Assuming all other parameters remain constant
parameters, it can extended easily to support other possible features.

To effectively deal with the adaptive load balancing problem, we need to consider two important issues.

1) Automated optimal reducer count: Heuristics to calculate the optimal number of reducers.

2) Load Balancing: Dividing the data processing workload equally between the reducers.

In this section, we discuss some methods that can deal with the issues mentioned above. We analyse in detail each of the approaches, and list out the positive and negative aspects of each approach. We also discuss the engineering implications of each design.

III. FLUBBER

Flubber is a pre-job that can be sandwiched between the actual job submitted by the user and Hadoop, the MapReduce framework. Interposing between the two Flubber creates a slice of the original job and analyzes the mappers’ behavior with this sample of input. This allows us to gather valuable information on the mappers’ output data distribution. Exploiting this information in hand, Flubber can go back to the original job and help it to retune/set some of the configuration parameters. For now, Flubber addresses the issue of balancing load across the reducers by trying to identify the ideal number of reducers for a given job.

A. A pre-job for statistical analysis

Flubber’s (or the pre-job’s) job configuration is identical to the actual user job, except in the input size and the reducer processing. As the pre-job is primarily used for statistics estimation, a part of the actual input will suffice, assuming the sample considered summarizes the entire input. To ensure this, we design a random text input format that randomly picks up a sample of records from the input dataset. The size of the sample, $\sigma$ relative to the input size, is specified by the user.

After the mappers are ran on the sample input, the next step is to collect the statistics from the output of the mappers. The simplest way of doing this would be to retrieve the size of the map output (stored in the counters class - org.apache.hadoop.mapreduce.counters). To get the optimal number of reducers($|R|$) would be

$$|R| = \left\lceil \frac{\text{Map output Bytes}}{\sigma \times \text{fs.inmemory.size.mb}} \right\rceil$$

The rationale behind the above formulation is as follows. Ideally, we want to divide the mapper output in such a way that the reducer buffer does not spill any records during merge phase. Assuming the output of the mapper is proportional to the size of the input, the output size for the entire input data will be $\frac{1}{\sigma}$ of the sample map output. This value needs to be shared in such a way that the amount of data each reducer processes is less than the reducer buffer size (specified by fs.inmemory.size.mb). Thus, we end up with the above formulation.

The heuristic above assumes that all the keys have equal number of values. But, the calculation above can be affected by a skewed distribution. For example, consider the following example.

Example 1: Assume $\sigma = 1$, i.e., sample input is the same as the actual input file. Let there be 11 keys in the map output. Assume that the size of the data associated with key 1 is about 10000 bytes. Here, size if the sum of the number of bytes for all $<\text{Key},\text{Value}>$ pairs having a particular key. Also lets assume that all other keys have about 1000 bytes of data each. If the reducer buffer size is 2000, the optimal number of reducers would be 6, i.e., allocate 1 reducer for the key 1 and allocate 5 reducers for the other 10 keys (2 keys per reducer). But, the formulation above will determine that the optimal number of reducers is actually 10. This number does not let us maximise the use of each reducer’s capacity. In addition, this will lead us to believe that the reducer that processes key 1 is a straggler node.

To overcome this negative effect, we assume that one reducer is allotted to a key whose projected size is greater than the reducer memory buffer size. For all the other keys, we calculate the ideal number of reducers required for processing these keys. Let $S$ be the list of all keys in the sample. Let $K$ be the set of all the keys such that their projected value over the entire dataset is $\frac{\text{size}(k_i)}{\sigma} > \text{fs.inmemory.size.mb}$, where $k_i$ is a key in the set $K$. The modified formula for finding the number of reducers is as follows.

$$|R| = |K| + \left\lceil \frac{\sum_{k_i \in S-K} \text{size}(k_i)}{\sigma \times \text{fs.inmemory.size.mb}} \right\rceil$$

Also, we have to take into consideration the case in which the number of keys is less than the number of reducers computed by the above formulation. This leads us to the formulation,

$$\text{Optimal number of reducers} = \min\left(\frac{|\text{keys}|}{\sigma}, |\text{reducers}|\right)$$

Once, this computation is done, we initialize the number of reducers for the actual job and the actual user given job continues with the optimal number of reducers.

This formulation gives us the number of reducers, but it does not make sure that the load is balanced across all the reducers. For this, we need to create a partition mapping, where each key that has a lot of data associated with it is assigned one reducer and all the smaller keys and the keys that are not considered in the sample are assigned to one of the remaining reducers. This way, we can make sure that the keys with large amounts of data don’t get mixed up with the smaller keys.

IV. RESULTS

We have tested our Flubber implementation on different sets of data and on different cluster configurations such that we can obtain a good overview of its performance.

The number of reducers has been varied in order to be able to compare the running time with the number of reducers as
computed by Flubber and the running time with the number of reducers based on the known heuristics [5].

In Figure 1 we can see the running time of the WordCount application for 2GB of input data, on the local hadoop21.cs.duke.edu cluster. The optimal number of reduce tasks, as computed by Flubber was 27, very close to the $0.95 \times \text{number-of-reduce-slots} = 28$. After running the WC application several times we can observe that, even if the running time varies with a certain degree from one instance to another, the time taken to compute using 27 reducers was approximately the best. Unfortunately, when adding the running time of computing the number and reducers and the key-reducer mapping we can see a decrease in overall performance.

![Fig. 1. Running times for 2GB of input on the 15 nodes cluster hadoop21.](image1)

The running time for the computation of WordCount using 216 reducers is higher than the running times using the values indicated by heuristics, the difference is not very large.

Because the current results are not very conclusive we shall run other experiments using other applications and datasets.

V. FUTURE WORK

While the approach we have considered as part of our project is good in a software engineering perspective, it does not give us a control over the partitioning done in the mappers. So, we cannot be sure of the load being balanced even with the ideal number of reducers. So, as a future direction of this project, we can compare the performance of the Flubber job with the approach explained in this section.

A. Partition Tuning approach

In this approach, we start with a pre-decided number of reducers (a really large number) and partition the output of the mappers based on this number of reducers. At a certain point, for example, when the mappers have completed about $\sigma$ of the processing, we tune the partitions based on the condition that none of them have more than $\sigma$ of the size of the reducer. We believe the resulting number of reducers will be optimal for processing the job.

A detailed description of each stage is given below.

1) Initial number of reducers: The initial number of reducers can be set either dynamically or default or given by the user.

As the default number is 1, there would be a lot of tuning done to end up with the ideal number of partitions. So, this configuration is not recommended if the job needs to process huge amounts of data (which is the case, typically).

For setting the number of reducers dynamically, we can consider the following approach to minimize the tuning of the partitions.

- We assume that the size of the output of the mapper is proportional to the input of the mapper. The number of reducers can be calculated simply as the size of input data / $\text{fs.inmemorysize.mb}$.  
  **Advantage:** No extra overhead to decide the initial number of reducers. It is straightforward and fast.
  **Pitfall:** The basic assumption may be wrong, in which case there will be a lot of partition tuning to be done in the later stages.

It is recommended that we take a really large number of initial partitions (also called virtual reducers). This will allow us to only focus on merging the partitions rather than splitting the partitions, which is more cumbersome and will affect the running time of the job.

2) Partition Tuning: For partition tuning, we use the bucket tuning idea used in dynamic hashing. The only difference is that the buckets need to be tuned across a cluster of mappers and the tuning needs to be done considering the data output by each of the mappers. To tune the partitions across all the mappers, we plan to take a screenshot of the data generated after some percentage of the mapper task is done. A typical
screenshot will contain the partition numbers and amount of data associated with each partition. This will be sent to the job tracker. The job tracker decides the partitions that need to be merged to give us the optimal number of partitions. These partitions need not be explicitly merged. Rather, we map a group of partitions for each reducer. This way, we can continue using the same partitions as above while also decreasing the number of reducers used.

The positives of this approach is that we have a better control over the partitioning of the output of the mapper. But, this approach involves a considerable modification of the hadoop code, which is negative in an engineering perspective.

VI. RELATED WORK

As MapReduce is a fairly new framework, little work has yet been done in ensuring adaptive load balancing. One approach has been the idea of dynamic profiling through sampling of the input data of the mappers. Considering that for most of the MapReduce applications the data distribution of the input of mappers is not the same as for the output, we try to improve upon this by eliminating the dependence of the number of reducers and the input data. [1] Another approach has been the load stealing by a reducer task, from another reducer task that is slower due to its surplus of work. [2] [4] Unfortunately, this represents just the splitting of equal amounts of workload, but deciding the number of reducers is still a parameter that has to be set by the user. We consider this as a possible improvement of our solution.

VII. CONCLUSIONS

With Flubber we have presented an approach to balance the load of MapReduce job by balancing the load across the reducers in the cluster. Flubber also demonstrates the feasibility and effectiveness of building systems to help users with job configuration and submission on top of MapReduce frameworks like Hadoop. Our current focus is on improving the reliability of Flubber without affecting adversely the running time of the job and providing strong guarantees about the impact on the running times. We are also working on adding support to compute based on the map outputs other job configuration parameters of interest.

REFERENCES