Towards Improved Load Balancing for Data Intensive Distributed Computing

Sven Groot
Institute of Industrial Science,
The University of Tokyo
4-6-1 Komaba Meguro-ku,
Tokyo 153-8505, Japan
sgroot@tkl.iis.u-tokyo.ac.jp

Kazuo Goda
Institute of Industrial Science,
The University of Tokyo
4-6-1 Komaba Meguro-ku,
Tokyo 153-8505, Japan
kgoda@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
Institute of Industrial Science,
The University of Tokyo
4-6-1 Komaba Meguro-ku,
Tokyo 153-8505, Japan
kitsure@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Specialized frameworks for highly scalable data processing continue to gain prominence over traditional databases in many environments including the cloud. Perhaps the most well-known such framework is Google MapReduce, which has gained wide-spread popularity. However, the MapReduce model offers some significant challenges for workload balancing which have not been adequately explored so far. In this paper, we introduce techniques for improving load balancing – particularly multi-stage jobs and dynamic partition assignment – by using a modified programming model that offers greater flexibility but maintains the simplicity, scalability and fault tolerance of MapReduce. We then explore the effectiveness of our approach using a parallel frequent itemset mining algorithm.

1. INTRODUCTION

Data analysis is the cornerstone of many business and research ventures. The amount of data to be analyzed keeps growing due to the web, and recently other sources such as sensors also contribute to this.

Google MapReduce [6][7] provides a framework for distributed data processing offering very high levels of scalability and fault tolerance. By building on the redundant storage provided by the Google File System [8] and by splitting jobs into small pieces that can be automatically re-executed in the case of failures, it can deal with scenarios like hardware failure more easily than traditional database systems.

Thanks to Hadoop [3], the MapReduce model has found widespread adoption even outside of Google. Hadoop has been adopted by many large players in the industry including Yahoo, Facebook, Amazon, etc., and has a rich ecosystem of applications built on top of it.

Cloud computing is a natural environment for these types of data processing systems. The cloud allows dynamic provisioning of resources based on your needs, and because using 1 node for 100 hours typically costs the same as 100 nodes for 1 hours, using the high level of scalability that MapReduce provides gives you greatly increased performance at no additional cost. MapReduce has been embraced by several cloud vendors; for example, Amazon provides Elastic MapReduce [2] as a platform for running dynamically provisioned Hadoop application on their EC2 and S3 platforms.

Part of the reason why MapReduce is so popular is its simplicity. It is easy to program for even without using higher level abstractions, and the users can understand the data flow of their applications quite easily. The simplicity of its model is also largely responsible for its scalability.

However, the MapReduce model is by its very nature quite inflexible, which makes it ill-suited for some types of applications. It has been shown that MapReduce performance can lag behind that of traditional parallel databases for some applications [15]. Yet because of its widespread usage, it is now being used for such applications anyway. And despite its apparent simplicity, it can also be very difficult to properly configure MapReduce applications for optimal performance, which is especially difficult in the cloud where resources are dynamically provisioned [5][13].

One important factor in this is workload balancing. If some of the nodes are not fully utilized – for instance because they are held up waiting for other nodes – this increases the overall execution time of the job. This can be caused by heterogeneous hardware; a node might be slower than other nodes because it has a slower CPU, disk or network connection. In addition, data placement imbalance can make it difficult to utilize some nodes because data needs to be moved before they can process it, and data skew may cause some tasks to have higher processing times than others. Other applications running on the nodes and factors such as disk fragmentation levels can also affect the relative speed of a node.

Many of these factors can not be determined in advance, and the uncertainness of dynamically provisioned resources in a cloud environment makes it harder to predict the load balancing behavior of an application. Therefore, it is important to be able to respond to changes in the load dynamically.

To our knowledge, there has been very little existing work studying the workload balancing characteristics of MapReduce. General load balancing issues in MapReduce are examined in [10]. The effectiveness of speculative execution – a simple mechanism employed by Hadoop where the same

SAC'11 March 21-25, 2011, TaiChung, Taiwan.
Copyright 2011 ACM 978-1-4503-0113-8/11/03 ...$10.00.

task is scheduled on multiple nodes – in heterogeneous environments has been studied in [19], and in [18], scheduling map tasks for load balancing was investigated.

There is a large body of work about load balancing in distributed systems in general. An overview of load balancing techniques in the cloud is given in [16]. Other work has focused on run-time load balancing on more traditional heterogeneous clusters [17][9]. However, the applicability of these kinds of techniques to MapReduce style applications has not been investigated.

In this paper we describe the load balancing issues that are present in MapReduce, and propose techniques to address some of these issues, particularly the use of multi-stage jobs and dynamic partition assignment. We evaluate these techniques using Jumbo, our own data processing platform, which strives to maintain the good points of MapReduce – simplicity, scalability, and fault tolerance – while mitigating some of the bad points so it can more effectively accomplish load balancing.

We will then use Parallel FP Growth [14] as an example to demonstrate how these methods can help improve load balancing and overall performance. Frequent itemset mining is an often-used data mining task which has many different applications in many different fields. In addition, this algorithm clearly demonstrates some of the shortcomings of MapReduce. The problems, and our solutions, are however not limited to this algorithm and are generally applicable.

2. WORKLOAD BALANCING

The MapReduce model presents several unique challenges for workload balancing that are caused by the basic design of the programming and execution model. Separate load balancing issues are presented by the map and reduce phases, and by the issue of complex algorithms.

2.1 Map Phase

The map phase consists of a large number of tasks, each of which processes a shard of the input data. Because this number is typically far larger than the number of map tasks that the cluster can run simultaneously the map phase is somewhat self-balancing, since faster nodes will simply run more tasks.

However, map tasks can still have load balancing issues. Some imbalance can occur at the end of the map phase when there are few tasks left; the severity of this depends on how much work is done by each map task.

Resources that are underutilized due to imbalance at the end of the map phase cannot be effectively used by reduce tasks. This is because the programming model for reduce tasks does not allow for incremental calculation of the result for multiple keys; the reduce function cannot be invoked until all values for at least one key are known, which is not the case until all map task inputs have been copied. Reduce tasks are therefore limited to copying intermediate data and running additional merge passes while the map phase is in progress.

For optimal performance, map tasks are scheduled according to data locality; the tasks are assigned to nodes that have a local copy of their input blocks. However, if there is data imbalance on the DFS some tasks will have to be non-local which may make it impossible to fully utilize the resources of some nodes. Additionally, non-local tasks incur disk and network I/O overhead, not only on the node where they are running but also on the node that provides the data.

2.2 Reduce Phase

Unlike the map phase, the number of tasks in the reduce phase of a job typically equals or is less than the capacity of the cluster. This enables the tasks to copy intermediate data and run additional merge passes in the background while the map phase is still running, and also helps to limit the number of network transfers. Using slightly less than the capacity is done so that the reduce phase can finish in one "wave" even if some nodes fail.

Increasing the number of tasks beyond the cluster's capacity is not recommended. Each task brings with it overhead due to initialization, increased network transfers, and merging the tasks' inputs. And because the second wave of reduce tasks cannot start until after the map phase has finished, they cannot copy intermediate data or do any work during the map phase, unlike the first wave.

Because of the small number of tasks, the reduce phase does not have the self-balancing nature of the map phase; when a node finishes a reduce task, there will be no more tasks from that job it can run. If one node is faster in completing the reduce tasks, it will not be utilized for the remainder of that job. The more time the job spends in the reduce phase after the completion of the map phase, the more pronounced this difference will be.

Although increasing the number of reduce tasks may improve load balancing, the overhead from using additional tasks means this will likely decrease the overall efficiency of the job and increase the execution time.

In order to balance the reduce phase we must attempt to assign more work to faster nodes. If the number of tasks does not exceed the cluster's capacity, how much data each node receives depends on the partitioning function and on how many reduce tasks it can run in parallel. The partitioning function does not know which task is assigned to which node so it is not a good mechanism for this.

Although running more tasks in parallel could be used to assign more work to a node, this can cause resource contention and actually reduce the effective utilization of that node. This is also a very coarse-grained mechanism and is only suited for static load balancing.

Data skew can be a major factor in load balancing issues in reduce tasks; if the partitioning function does not evenly distribute the work across each task, they can have greatly varying execution times.

2.3 Complex Algorithms

Another major issue is caused by MapReduce's inflexible model. Many complex algorithms do not fit exactly into the structure of one map and one reduce phase, and will require more than one MapReduce job.

If more than one job is used, this means that intermediate data between the jobs has to be stored on the distributed file system. This data is often replicated, creating additional overhead. Although it's possible to disable replication this will in turn make it harder to schedule data local map tasks in the next job.

A MapReduce job cannot be submitted unless all its input data is available. The subsequent jobs depend on data from the previous one, and can therefore not be started until the previous one completes. There is no opportunity to start work with partial data to employ available resources. This means that if one job is not properly load balanced, this prevents the faster nodes from continuing with the next job until the slower ones can finish working on the current one, causing the effects of workload imbalance to accumulate.

Additionally, the scheduler is only aware of a single of these jobs at a time, since the next one has to wait until all data from the current one is available until it can be submitted. The scheduler does not know the overall structure of the application, and therefore cannot make more intelligent load balancing decisions. This also hinders the ability for the scheduler to adhere to execution policies like job priority if more than one job can be active in the system at the same time.

3. JUMBO

In order to implement and experiment with load balancing techniques we have developed Jumbo, a highly scalable distributed data processing system in the spirit of frameworks such as MapReduce or Microsoft Dryad [12]. Jumbo maintains many of the properties of MapReduce and will be easy to use to anyone who has used MapReduce. However, Jumbo offers a more flexible programming model that allows us to deviate from MapReduce's limitations. While more flexible, Jumbo's data flow is closer to MapReduce than Dryad's, making it somewhat easier to reason about the structure of an application.

We will give an overview of Jumbo's basic design, and then describe the load balancing techniques it uses to mitigate some of the issues presented in Sect. 2.

3.1 Overview

Jumbo consists of two components, a distributed file system and a distributed data processing environment.

The Jumbo DFS operates on principles that are very close those of GFS and the Hadoop DFS. Data is divided into large blocks which are replicated on the cluster. A single name server is responsible for managing the file system name space.

Jumbo Jet is the distributed data processing platform for Jumbo. It provides a programming model and an execution environment which aims to maintain the scalability and fault tolerance of MapReduce, but at the same time allow for greater flexibility while sacrificing as little of the simplicity as possible.

Jumbo maintains some of the properties of MapReduce. The MapReduce model provides several basic features: it provides parallelization through data sharding and partitioning, and fault tolerance by re-execution of failed tasks and the materialization of intermediate data in files. It also provides a programming model based on key/value pairs that are processed by map and reduce functions where the intermediate data between the two is grouped by key, and it provides an implementation of that model that uses sorting to to achieve the grouping.

Jumbo provides the same parallelization and fault tolerance features, but it is more flexible with regards to the model. Records don't need to be key/value pairs, it is not required to do grouping, there is no forced distinction between map and reduce tasks, and there is no requirement to use sorting. Jumbo still can do all those things, but also allows the use of alternatives where appropriate.

A data processing job in Jumbo consists of one or more *stages*. Each stage is divided up into multiple tasks which do the same operation but on a different fragment of the input data. Tasks are the unit of work used by the scheduler, and scheduling these to run on different nodes is how parallelism is achieved, similar to how MapReduce works. If a task fails it can be automatically re-executed without failing the entire job.

Stages are connected via *channels* that represent data flow. A stage reads data either from the DFS or from one or more channels, and writes output to the DFS or to a channel. Input from the DFS is split into pieces (typically using DFS blocks) and channels use partitioning to split data across tasks. Stages that read data from the DFS can be compared to map phases, and stages that read data from a channel can be compared to reduce phases, although there is no requirement that they perform equivalent operations. However, they frequently display similar characteristics when it comes to load balancing and performance.

Jumbo materializes intermediate data for channels in files on the local disk which are transferred over the network for maximum reliability and to allow virtualization of the cluster's resources.

A channel contains all the configuration related to the intermediate data. It indicates how the data is partitioned, and how the data from multiple tasks in the stage that writes data to the channel is combined by the stage that reads data from the channel. Unlike in MapReduce, it is not required that intermediate data is sorted. Sorting is available if required, but it is also possible to use alternative methods of grouping, or not group the data at all. Built-in functionality is provided for the most common operations (including sorting and hash table based grouping), or custom functionality can be used if none of the built-in options suit the application.

Jumbo uses partitioning to divide data across tasks that read from a channel in a way that's similar to a reduce phase in Hadoop. However, in MapReduce, partitions are coupled to reduce tasks; each reduce task processes exactly one partition. In Jumbo, it's possible to specify that a single task will process more than one partition. Because partitioning itself is a form of grouping, it's possible that this is the only grouping needed. By using multiple partitions per task, you can take advantage of that even if the number of groups (partitions) is relatively large, without needing to incur overhead from increasing the number of tasks. This provides a method of implicit grouping that does not depend on sorting.

Because Jumbo uses the same strategy of data sharding and partitioning for parallelization, it has nearly the same scalability characteristics as MapReduce, and also the same fault tolerance (as long as the TCP channel type is not used). This also leads to jobs that, in practice, have a structure that will look similar to their MapReduce equivalent, and is similarly easy to understand. Although some of the flexibility, like whether to sort and what grouping strategy to use, requires more understanding of distributed systems than MapReduce, you can always just pick the safe defaults and get similar results as in MapReduce. In the future, we hope to let the runtime system make some of these choices for the user to further increase its ease of use.

Job execution in Jumbo is handled in a similar manner as in MapReduce. A single server called the job server schedules tasks to nodes in the cluster and keeps track of their status. If a task fails or a server becomes unavailable, tasks can be re-executed elsewhere to ensure fault tolerance.

3.2 Multi-Stage Jobs

In Sect. 2.3 we described how load balancing issues can occur when multiple jobs are needed for one algorithm. Our first load balancing technique uses the flexibility of Jumbo, which allows us to create more complex job graphs with many stages and eliminate the need for multiple jobs.

We are able to use this approach with Jumbo because a job in Jumbo is not limited to one map and reduce phase. A job can have any number of stages, with each stage reading output from the previous stage. It is also possible to have more than one stage that reads data from the DFS in a single job. This makes it possible to represent complex algorithms in a single job, rather than requiring multiple jobs as would have been necessary in MapReduce. The use of stages connected by channels means that, in practice, these job graphs will closely resemble the equivalent sequence of MapReduce jobs, which makes it easy to understand and use, and maintains the scalability of MapReduce.

By using a job with multiple stages, it becomes possible to schedule tasks from a subsequent stage if one stage is not perfectly load balanced. In MapReduce frameworks such as Hadoop, the map and reduce phases are running simultaneously, but with multi-stage jobs it becomes possible to run stages simultaneously that in MapReduce would not have been part of the same job.

The current Jumbo scheduler will create a dependency graph of all stages in the job and schedule tasks from those stages in that order as capacity becomes available (while also attempting to satisfy data locality for tasks that read from the DFS). Even this relatively simple strategy allows for overlapping execution of stages which in MapReduce would be in a different job. We expect to improve the effectiveness of this as we continue to improve the scheduler.

The effectiveness of running tasks from multiple stages simultaneously depends on the ability of the tasks to start doing work even though not all input data is available yet. In MapReduce, reduce tasks can copy intermediate data and run additional merge passes if needed. However, because the programming model for reduce functions does not allow incremental calculation and because MapReduce uses sorting of the intermediate data to group the keys, it cannot do much else.

Jumbo has the ability to use alternative grouping methods and its programming model also allows for incremental calculation of a group's results (which can for example be used for many aggregation functions). This makes it possible to start running the core function of the task even with partial data. Depending on the structure of the data and the type of operation being used, Jumbo can do a large part of the work before the final piece of data becomes available, making it possible to use available resources on a node while other nodes are still processing tasks from the input stage.

This ability to do more work with partial data also applies to stages that would have been part of the same MapReduce job (as a map and reduce phase), which allows us to reduce the impact of imbalance at the end of a stage reading from the DFS.

3.3 Dynamic Partition Assignment

Our second load balancing technique is aimed at the issue with long running reduce tasks as described in Sect. 2.2. This kind of imbalance can also occur in Jumbo for a stage that reads from a channel, since they have roughly the same characteristics as a reduce phase in MapReduce.

In order to divide data between such tasks, it is partitioned. As mentioned in Sect. 3.1, Jumbo allows the use of more than one partition per task. Since a task can have many partitions, we can dynamically reassign these partitions to different tasks during execution to improve load balancing.

When a task starts, it is assigned a number of initial partitions. Currently, our strategy for initial partition assignment is to simply divide all the partitions evenly across all the tasks. A task will copy intermediate data for all of the partitions it was assigned. If processing using partial data is possible, this will be done only for the first partition; although it would be possible to process multiple partitions in parallel, currently this is not done to prevent resource contention.

A task will process its partitions sequentially. Before processing for a partition begins, the task contacts the job server to check if the partition has been reassigned, in which case the task will discard all data for that partition and continue.

Once a task finishes all previously assigned partitions, it will contact the job server to ask for additional partitions. If it receives additional partition assignments, it will copy the intermediate data for those partitions – because intermediate data is materialized, the overhead this causes is spread across all nodes, not just the node that had previously been assigned the partition – and then it will process them, and repeat these steps until no new partitions get assigned.

In order to find a suitable partition to reassign, we use for the task with the largest amount of remaining partitions and the lowest overall progress. Although this strategy appears to work well so far, we expect to be able to improve on it further in the future.

Although dynamic partition assignment appears similar to simply using more tasks, it has several advantages. Firstly, it does not suffer from any initialization overhead. A task will also copy the intermediate data for all the initially assigned partitions as it becomes available; by contrast, if more tasks are used they must copy their data when they are started as there is no opportunity to do so earlier.

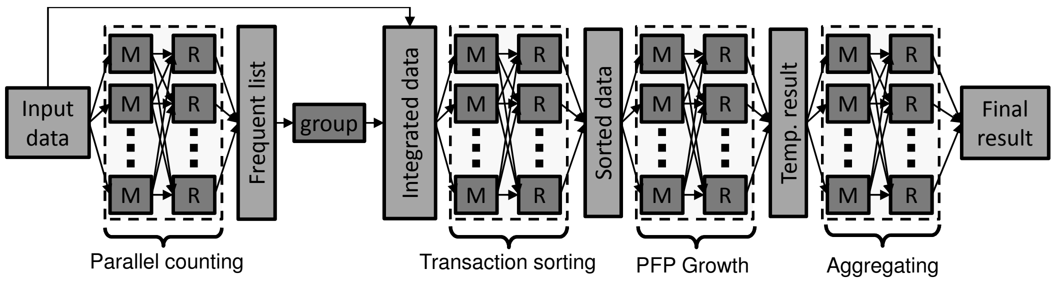
Of course, this means that some tasks will copy intermediate data for partitions that they may not process, leading to some unnecessary work. We expect to be able to minimize this overhead by improving the strategy for initial partition assignment. The data for dynamically reassigned partitions still needs to be copied before processing can begin. However, our experience shows that the overhead of doing this is still significantly smaller than using multiple tasks.

Dynamic partition assignment is a technique that could also be applied to MapReduce. Decoupling the partitions from tasks and implementing dynamic assignment does not lead to any fundamental changes in the model so it could be used by a MapReduce framework as well. However, our experience shows that it is harder to use this strategy effectively when the intermediate data needs to be sorted as is the case in Hadoop and Google's MapReduce, so it may be somewhat more difficult to implement in that situation.

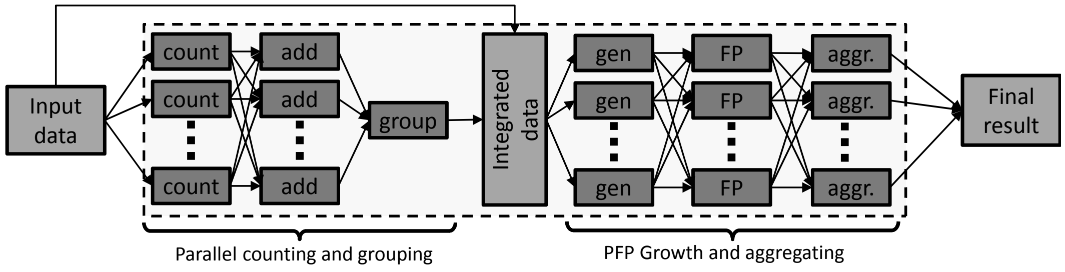
4. PARALLEL FP GROWTH

4.1 MapReduce

In order to evaluate load balancing issues on MapReduce and Jumbo we have used FP Growth [11] frequent item-



(a) MapReduce (Hadoop/Mahout)



(b) Jumbo

Figure 1: Job graphs for the Parallel FP Growth algorithm with Hadoop and Jumbo. The Hadoop version uses four jobs, while the Jumbo version has only one.

set mining as an example. FP Growth efficiently discovers frequent itemsets by constructing an FP tree from the transaction database and mining that instead of the original database.

For our experiments we used Parallel FP Growth [14], a parallel version of FP Growth used to discover the top-K frequent itemsets for each item in the database. It was designed by Google for use with MapReduce, and we adapted it for use with Jumbo so it can make use of the load balancing techniques described in Sect. 3.

PFP Growth in MapReduce was designed to avoid synchronization issues by partitioning the transaction database into groups. Each group contains all the data needed to do frequent itemset mining on the features in that group, so there is no need for synchronization. Because the result of each group can include patterns containing features that were not part of the group, these results must be aggregated into the final result.

Mahout [4] is an open source machine learning library for Hadoop, and includes an implementation of PFP. In the remainder of this paper, we will use this implementation as our source of reference. It is mostly the same as the version described by Google, but has two important differences: it uses a special intermediate format to reduce the size of the intermediate data and the cost of FP tree construction in the reduce phase, and it has an extra step which sorts the entire transaction database.

The structure of Mahout's PFP Growth implementation is shown in Fig. 1(a). The algorithm consists of five steps, four of which are MapReduce jobs. The *parallel counting* step counts the number of occurrences of each feature in the transaction database. The *grouping* step sorts the list of items by frequency and divides it into groups; this is not parallelizable and takes only a few seconds, so it is not done with MapReduce. The *transaction sorting* step sorts each transaction by item frequency, and also groups the items based on their most frequent item. The *PFP Growth* step generates group-dependent transactions from each transaction in the map phase, and the reduce phase performs FP Growth on each group. Finally, the *aggregating* step aggregates the results of each group into the final answer.

There are a number of points where imbalance can occur. The obvious ones are between the jobs, but because reduce tasks can do only limited work while the map phase has not finished, there is also the opportunity for hold-ups here.

4.2 Jumbo

When porting the PFP Growth algorithm to Jumbo, we first identified how to collapse the algorithm into a single job. The result of this is one job with six stages, shown in Fig. 1(b).

The *count* and *add* stages count item frequencies. The channel between these stages does not sort the data; instead items are grouped in a hash table and the counts are incrementally updated, which is much more efficient. The *group* stage is still not parallel (it has only a single task) but it is now part of the job. The *gen* stage sorts each transaction by item frequency, and then generates group dependent transactions. The *FP* stage performs FP Growth on each group. The *aggregating* stage finally aggregates the results; note we eliminated the map phase of this step, because it only served to repartition the data, something which the channel between the *FP* and *aggregating* stages now does.

Having the algorithm in a single job, rather than four as in MapReduce, allows us to be more intelligent about scheduling. Although the *gen* stage must still wait for grouping to be complete before it can do anything useful, all of the remaining stages can do some work with only partial data available from their input channel, allowing these stages to utilize nodes that have no more tasks from the previous stage to run.

In the *PFP Growth* MapReduce job, sorting was used to group the intermediate transactions by group ID. Since the number of groups is usually relatively small (typically several thousand) we chose to match the number of partitions to the number of groups, using the multiple partitions per task feature to still assign multiple groups to each task. Each partition gets exactly one group, eliminating the need for additional sorting or grouping. Because no sorting is used, the *FP* stage can already begin building FP trees with partial data. Because the number of groups is typically larger than the number of tasks, we can then also use dynamic partition assignment for load balancing.

The *aggregation* stage receives results for a linear range of group IDs, and each group's results are typically small (depending on the value of K) and can be computed incrementally as new temporary data arrives. Therefore, these tasks simply store the groups in an array indexed by group ID, again eliminating the need to sort and making it possible for them to work with partial data.

It should be noted that PFP Growth in Jumbo still follows roughly the same structure as in MapReduce. Porting it was fairly trivial; all we did was combine some steps and choose different grouping strategies so we could eliminate sorting. Implementing it was not any more difficult than it was for MapReduce, and in fact, since we don't need to make allowances to fit to MapReduce such as the extra map phase for aggregation, in some ways it was actually simpler.

5. EXPERIMENTAL RESULTS

We evaluated PFP Growth on Hadoop and Jumbo both for performance and load balancing. For these experiments we used a cluster with 48 nodes; of these, 32 were *type A* nodes with an Intel Core 2 Duo E6400 CPU (a total of 2 cores), 4GB RAM, and a single 1TB disk. The remaining 16 were *type B* nodes with two Intel Xeon E5410 quad-core CPUs (a total of 8 cores), 32GB RAM, and two 1TB disks. Each node runs as many tasks in parallel as it has CPU cores.

We used a synthetically generated database [1] with between 100 and 800 million transactions, with sizes of 5.5GB to 44GB respectively. Each database had 100,000 unique items and an average transaction length of 10 items. We used a minimum support of 0.001%, which resulted in around 25,000 frequent items in all cases.

We separately investigated the effects of both dynamic partition assignment and multi-stage jobs.

5.1 Dynamic Partition Assignment

To determine the effectiveness of dynamic partition assignment, we ran PFP growth with four different workloads on Hadoop, Jumbo without dynamic partition assignment, and Jumbo with dynamic partition assignment. The results are shown in Fig. 2. The configuration of Jumbo and Hadoop was tuned separately for each job to attempt to provide optimal performance. Dynamic partition assignment was applied only to the *FP* stage of the job, because it is the longest stage and the only place where a significant amount of imbalance could occur.

The first thing to notice is that Jumbo is on average around 3 times faster Hadoop. This is primarily because

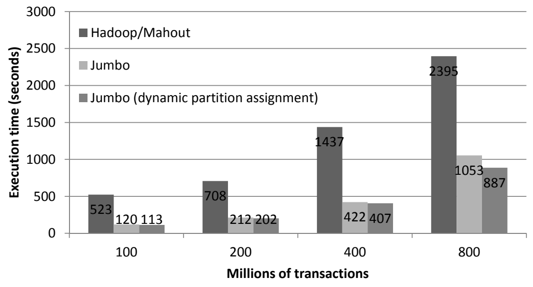


Figure 2: Execution times of PFP Growth on Hadoop and Jumbo with and without dynamic partition assignment.

Jumbo's greater flexibility allowed us to eliminate sorting of the intermediate data and use more efficient grouping methods, which reduces a very large amount of overhead. Part of this speed-up is also due to the use of multi-stage jobs. Because the entire application is only a single job, there is no intermediate data stored on the DFS, and there is increased opportunity for tasks to work in the background with partial data. This will be explored further in the next section.

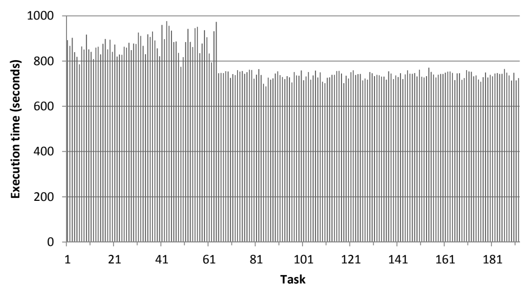
When looking at the effectiveness of dynamic partition assignment, it appears to give only a small benefit except for the biggest workload, where it is approximately 16% faster than Jumbo without dynamic partition assignment. This is because the grouping is effective in ensuring each task has about the same amount of work. The FP Growth algorithm is largely CPU bound so the CPU speed determines execution time. Each individual core on the type A and B nodes is close to the same speed; the type B nodes have more cores, but they can be used by running more tasks in parallel. This means that there is not much imbalance to begin with, which limits the maximum theoretical gain.

With 800 million transactions, the intermediate data becomes large enough to create some I/O overhead on the type A nodes, making the speed difference between them larger. This means there is more imbalance and therefore more improvement when using dynamic partition assignment. We expect that when data skew and other factors to cause imbalance are introduced, we will be able to show a greater gain.

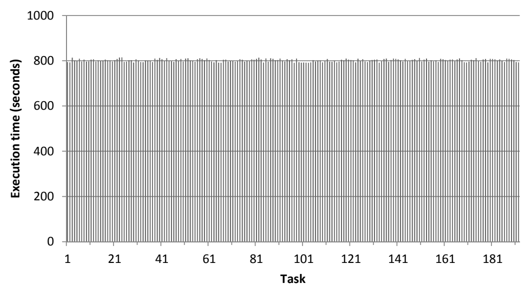
We also confirmed that dynamic partition assignment improved load balancing by looking at the execution times of the tasks in the *FP* stage where the balancing was applied, as shown in Fig. 3. Without load balancing, the tasks running on the type A nodes clearly take longer than those on the type B nodes. There is also a fair amount of variance between the tasks on the type A nodes; this is because there is some disk contention on these nodes, which makes the overall execution time more unpredictable. The job cannot finish until the last task has finished.

When dynamic partition assignment is used, we can see that the tasks' execution times are now all nearly identical. The type B nodes take over some of the work from the type A nodes and are better utilized, leading to a faster execution of the job overall. The bigger the initial imbalance, the bigger the benefit of this approach will be.

We can also see this in Fig. 4, which shows the CPU usage of one of the type B nodes. In Fig. 4(b), dynamic partition assignment was not used. We can see that this node is idle

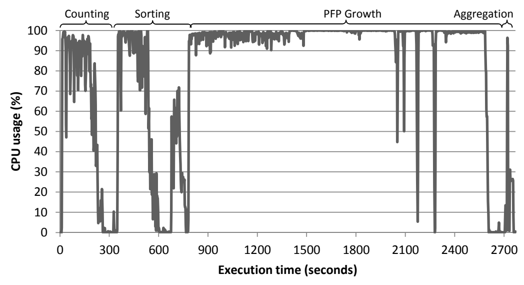


(a) No load balancing

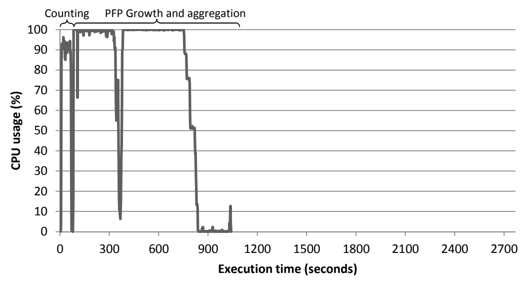


(b) Dynamic partition assignment

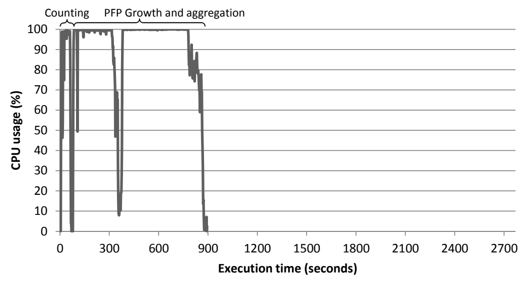
Figure 3: Execution times of the tasks in the *FP* stage using Jumbo. The first 64 tasks are executed on the type A nodes, the remaining 128 tasks on the type B nodes.



(a) Hadoop/Mahout



(b) Jumbo



(c) Jumbo (dynamic partition assignment)

Figure 4: CPU usage of one of the type B nodes.

for a long time near the end of the job while the other nodes are still working on their tasks.

In Fig. 4(c) we see the result of using dynamic partitioning. The node is now kept busy until the end of the job, which is true for all the type B nodes, causing the job to finish sooner. Full CPU utilization is not achieved due to the need for the tasks to copy the data of the additional partitions, but we still clearly achieve a much better utilization of all nodes than without dynamic partition assignment.

5.2 Multi-Stage Jobs

In order to show the effectiveness of using multi-stage jobs as described in Sect. 3.2 we cannot look only at execution time. Although part of the speed-up of PFP Growth in Jumbo over Hadoop is caused by the use of a multi-stage job, it is difficult to tell how much of the speed-up was the result of this, and how much was the result of other factors such as the more efficient grouping methods and other implementation issues.

Instead, we therefore look at node utilization. We examined CPU, disk, memory and network utilization for all of the nodes, particularly the type B nodes which are more likely to be underutilized because they are faster. Because we found that CPU was the limiting factor in most cases, we will only consider the CPU usage in the remainder of this discussion for the sake of simplicity.

Fig. 4 shows the CPU usage of one of the type B nodes for this job. Because the type B nodes are faster than the type A nodes, they are underutilized if load imbalance occurs, which shows in the graphs as low CPU usage. In these situations, there is no more work for the scheduler to assign to this node although other nodes in the cluster are still working on tasks in the same job.

In Fig. 4(a) we see the situation with Hadoop. This node is idle for some time during the end of the parallel counting job, the end of the map phase of the transaction sorting job (note that although CPU usage is less than 100% during the reduce phase of this job, this is because it is I/O bound, not due to imbalance), and the end of the PFP growth job.

Jumbo is shown in Fig. 4(b) and Fig. 4(c). We can see that the drop in CPU usage at the end of the counting step is much shorter. This is because both the *add* and the *group* stage could do work in the background with partial data, rather than having to wait. Because the *transaction sorting* job is not present in Jumbo, it can obviously not cause any imbalance.

Both Hadoop and Jumbo show a spike in CPU usage near the end for the aggregation tasks, but it is much smaller for Jumbo. In Jumbo, the aggregation tasks had already been running and were able to work with partial data. In Hadoop, the *aggregation* job takes a total of 45 seconds after the *PFP Growth* job completes. In Jumbo, the *aggregation* stage finishes in under 5 seconds after the final task of the *FP* stage finishes because it had already done most of the work.

Jumbo shows one drop that Hadoop does not, which occurs at the end of the *gen* stage (which corresponds to the map phase of the *PFP Growth* job). The drop is due to the final few tasks being finished on other nodes than the one pictured here. This situation actually also occurs with Hadoop; the lack of a similar drop in CPU usage in Fig. 4(a) is due to speculative execution. Although this keeps the node busy, some of its work was discarded. Speculative execution does not solve the load balancing problem in this scenario, it just hides the symptoms. Speculative execution was not used in Jumbo.

6. CONCLUSION

In this paper, we have shown that MapReduce applications can suffer from workload imbalance issues, some of which are caused by the properties of the MapReduce model itself. Jumbo, our experimental data processing platform, aims to overcome some of these issues while still maintaining the good points of MapReduce such as simplicity, scalability and fault tolerance. Creating applications for Jumbo is not more difficult for those who have experience with MapReduce, but through the use of several techniques large performance bottlenecks can be eliminated.

Jumbo employs several techniques to reduce load balancing issues found in MapReduce; in this paper we have focused on the construction of multi-stage jobs and the use of dynamic partition assignment. The results obtained with Parallel FP Growth indicate that these techniques can lead to a better utilization of all nodes in a heterogeneous cluster, and improve the overall throughput of the job. This will save time, energy, and money; the latter especially in a pay-per-use cloud environment. These techniques are however not limited to Parallel FP Growth; they can be applied to a broad range of MapReduce style data processing applications.

Dynamic partition assignment is likely also applicable to MapReduce. Although the requirement to sort the intermediate data would make this more difficult to implement, we expect that it may still offer benefits in a pure MapReduce environment.

There are still a large number of open issues in this area. Workload imbalance caused by data placement imbalance or data skew is currently not sufficiently investigated. Load balancing can also become more difficult if disk I/O or network bandwidth is the limiting factor, since dynamically moving data around will always incur overhead on those two resources. We can therefore have a situation where assigning more work to one node can cause resource contention on another node. In our future work, we intend to address these issues.

In addition, we intend to more closely investigate the possibilities for balancing resource usage. Currently, Jumbo only uses execution time as a rough metric for the throughput of a node. By measuring the usage of CPU, disk, memory and network during job execution we expect to be able to make more intelligent decisions about how to schedule tasks and move data.

7. ACKNOWLEDGMENT

This research is partially supported by the project "R&D on reliable and power-saving network control technologies for cloud computing" of the Ministry of Internal Affairs and Communications.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Quest synthetic data generator. IBM Almaden Research Center.

[2] Amazon. Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[3] Apache. Hadoop Core. `http://hadoop.apache.org/core`.

[4] Apache. Mahout. `http://mahout.apache.org/`.

[5] S. Babu. Towards automatic optimization of MapReduce programs. In *Proc. of SoCC*, pages 137–142, New York, NY, USA, 2010. ACM.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of OSDI*, pages 137–150, Berkeley, CA, USA, 2004. USENIX.

[7] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of SOSP*, pages 29–43, New York, NY, USA, 2003. ACM.

[9] K. Goda, T. Tamura, M. Oguchi, and M. Kitsuregawa. Run-time load balancing system on SAN-connected PC cluster for dynamic injection of CPU and disk resource – a case study of data mining application –. In *Proc. of DEXA*, pages 182–192. Springer, 2002.

[10] S. Groot, K. Goda, and M. Kitsuregawa. A study on workload imbalance issues in data intensive distributed computing. In *Proc. of DNIS*, pages 27–32. Springer, 2010.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, 2000.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[13] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proc. of HotCloud*, Berkeley, CA, USA, 2009. USENIX.

[14] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. PFP: Parallel FP-growth for query recommendation. In *RecSys*, pages 107–114, New York, NY, USA, 2008. ACM.

[15] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of SIGMOD*, pages 165–178, New York, NY, USA, 2009. ACM.

[16] M. Randles, D. Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *Proc. of WAINA*, pages 551–556, Los Alamitos, CA, USA, 2010. IEEE.

[17] M. Tamura and M. Kitsuregawa. Dynamic load balancing for parallel association rule mining on heterogenous pc cluster systems. In *Proc. of VLDB*, pages 162–173, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[18] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *Proc. of*

ICDE, pages 657–668, Los Alamitos, CA, USA, 2010. IEEE.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proc. of OSDI*, pages 29–42, Berkeley, CA, USA, 2008. USENIX.