Jumbo: Beyond MapReduce for Workload Balancing

Sven Groot
Supervised by Masaru Kitsuregawa
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

ABSTRACT

Over the past decade several frameworks such as Google MapReduce have been developed that allow data processing with unprecedented scale due to their high scalability and fault tolerance. However, these systems provide both new and existing challenges for workload balancing that have not yet been fully explored. The MapReduce model in particular has some inherent limitations when it comes to workload balancing. In this paper, we introduce Jumbo, a distributed data processing platform that allows us to go beyond MapReduce and work towards solving the load balancing issues.

1. INTRODUCTION

With the ever growing amounts of data that companies and research institutions have to process, there has been a growing need for scalable and fault tolerant systems that allow the processing of such large volumes of data. When processing terabytes or even petabytes on large clusters of commodity hardware, hardware failures are to be expected on a daily basis and traditional database systems don't offer the facilities required to cope with this situation, nor can they scale sufficiently.

Because of this, the last decade has seen a move away from the relational database model towards more specialized solutions to store and process large amounts of data. Perhaps the most well known data processing platform to emerge from this need is Google's MapReduce [2][3][4]. It provides a simple programming model and an environment to execute MapReduce programs on large clusters that automatically handles scheduling and fault tolerance. This is accomplished by using the redundant storage of the Google File System [5] and by providing automatic re-execution of small portions of the job in case of failures.

The MapReduce model has gained widespread adoption even outside of Google, largely thanks to the open source implementation provided by Hadoop [1]. But for all its advantages, MapReduce is a very inflexible model that was designed to solve the specific data processing needs of Google, and is now being used for things it was not designed to cope with. This has been explored in [10] which compares MapReduce to parallel databases.

One important aspect when doing processing on large clusters is the ability to absorb heterogeneity. When replacing failed components or expanding the capacity of a cluster it is nearly impossible to guarantee that all nodes in the cluster will have the same capabilities in terms of CPU, disk or network speed. In addition, it might be the case that a cluster is being used for other work that could be outside of the control of the data processing framework. In this case, we might find some nodes are performing more slowly than usual because they are busy with other tasks.

Cloud computing adds additional complexity to this problem. One issue is scale; MapReduce and similar batch processing systems are often used on larger clusters and with larger amounts of data than is common for more traditional parallel databases. The cloud is also meant to be flexible, with dynamic allocation of computing resources. This means that today you might be using a different set of computers than yesterday. We have also seen a trend of increasing use of virtualization, where one physical server might be running many virtual machines. It is therefore hard to determine what the performance characteristics of the machines will be.

It is up to the execution environment to make sure we can optimally use the capabilities of faster hardware, and to make sure that slow nodes – whether due to hardware or workload – do not unnecessarily delay the execution of the jobs. Unfortunately some of the properties of MapReduce can make this more complicated and Hadoop, the most popular MapReduce implementation, does not have many provisions for workload balancing at all.

To our knowledge, only limited work has been done on load balancing in MapReduce[1]. The effectiveness of speculative execution – a simple load balancing mechanism employed by Hadoop – in heterogeneous environments has been studied in [13]. In [12], scheduling map tasks for load balancing was investigated.

In this paper, we introduce an approach to data processing that aims to keep the advantages of MapReduce in scalabil-

[1]Microsoft Dryad [8] is an alternative to MapReduce and offers a much more flexible programming model which can alleviate some of these concerns. But due to the pervasiveness of MapReduce and the limited availability (to our knowledge) of data concerning Dryad's functioning in heterogeneous environments we have not included it in the comparisons provided in this paper.

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

ity, fault tolerance and ease of use, but that goes beyond the MapReduce model so it can better address the load balancing challenges provided by that model. This is an elaboration on the work published in [7].

In the following sections, we will first discuss workload balancing in MapReduce, and then introduce our own data processing system, Jumbo, that is intended to address the issues presented.

2. WORKLOAD BALANCING

In this section, we will look at some of the issues for workload balancing that can arise in MapReduce in particular. Hadoop is used as the reference implementation for MapReduce in this discussion.

The MapReduce model divides a job into a map phase and a reduce phase. Each of these phases provides its own issues for workload balancing.

2.1 Map Phase

A typical MapReduce job has more map tasks than can be run simultaneously on the cluster. For example, a job using 1TB of input data with a 128MB block size would have at least 8192 map tasks, and the number can easily be increased even further.

Hadoop uses a greedy scheduling algorithm for these types of tasks, which means that each node will be assigned a new task as soon as it finishes the last one. As a result, nodes that are slower at processing map tasks will automatically run fewer tasks than faster nodes. In this sense, the map phase is self-balancing.

However, map tasks can still have load balancing issues. The Hadoop scheduler attempts to schedule map tasks on nodes that have the relevant input data, but when a node no longer has any data for any of the remaining map tasks, it will be assigned a non-local task. In this case, the node has to read data from another node, and network transfer or disk speed at the source node may limit the execution time. This might leave the processing power available on that node not fully utilized, because it is waiting for data.

The placement algorithm for DFS blocks and the scheduler's ability to minimize the number of nodes that need to run tasks against non-local data are therefore important attributes when load balancing map tasks. Currently, Hadoop's data placement algorithm is random and doesn't try to optimize data placement for job execution.

This issue is likely to come up in cloud environments. One of the properties of the cloud is that you can easily increase your capacity to meet increased demand. If you were normally using one hundred nodes, but today have scaled up to one thousand, most of those nodes do not have any data, and cannot be optimally utilized until they do.

Another issue can occur if one of the nodes is unexpectedly much slower in processing certain map tasks. If this happens to the last remaining map tasks, this can hold the entire job up, as the reduce phase cannot start until all map tasks are completed. The mechanism Hadoop uses to prevent this is speculative execution: when there are no more map tasks to schedule, Hadoop will execute already running map tasks on multiple nodes in the hopes they can finish them sooner. While this works reasonably well, it does lead to duplication of work and is therefore not an optimal strategy.

2.2 Reduce Phase

Contrary to the map phase, a job typically has only as many reduce tasks as the cluster can execute simultaneously, or slightly less. While it is possible to use more reduce tasks, this is not recommended. Every reduce task needs data from all map tasks, and can therefore not finish until all map tasks have finished. Reduce workers will shuffle and merge the intermediate data in the background while the map tasks are still running, so that all data is available soon after the last map task finishes. If there are more reduce tasks than the cluster can simultaneously execute, some of these tasks will not be started until after the previous reduce tasks finish. These tasks therefore do not have the ability to do any processing during the map phase, so increasing the number of reduce tasks beyond the cluster's capacity will increase the execution time of the job.

Because there are no more tasks than the cluster can run, reduce tasks do not have the self-balancing nature of map tasks. Once a node completes its reduce tasks, there will be no more work for it to do for this job. If one node finishes faster (for example because it has faster CPUs or disks, or because it has less other work to do), it cannot take any of the work from the other nodes. As such, the slowest reduce task determines the execution time of the entire job.

When the intermediate data is large, reduce tasks can make up a large part of the execution time of the job. If the input data to a single reduce task does not fit in memory (which is common), it needs to perform an expensive external merge sort. This is a disk intensive process, so nodes with faster or more disks can finish considerably sooner.

The only way to balance the reduce tasks is to attempt to assign more data to faster nodes. How much data each nod receives depends purely 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 ill-suited to perform load-balancing.

Although you can assign more work to a node by having it run more reduce tasks in parallel, this is a very coarse-grained mechanism, and running more tasks in parallel on a single node can have negative effects when they are contending for the same resources such as disks. This approach is also only suited for static load-balancing, as the number of reduce tasks for a single node cannot be adjusted during job execution.

2.3 Complex Algorithms

The MapReduce programming model is very inflexible. It imposes a rigid structure that your job must adhere to, and if your algorithm does not follow this model exactly, you will often have to work around the limitations. In practice, this means that many complex algorithms will need more than one MapReduce job.

The necessity to have multiple jobs has several disadvantages. Intermediate data between the jobs has to be saved on the distributed file system which can cause unnecessary overhead. Because it is not possible for a MapReduce job to start until all its input data is available, each following MapReduce job cannot be started until the previous one has completely finished.

Additionally, the scheduler is only aware of a single of these jobs at a time. It does not know the overall structure of the algorithm, and has only limited information with which it can make scheduling decisions.

3. JUMBO

In order to evaluate workload balancing and other issues in data intensive distributed computing, we have developed Jumbo, a flexible data processing environment in the spirit of MapReduce and Dryad. The aim of Jumbo was to provide a system that maintains the scalability and fault tolerance aspects of MapReduce, but is more flexible so we can investigate alternative approaches that would not be possible with the MapReduce model.

We have decided to develop our own solution, rather than build on Hadoop, because some of the workload balancing issues outlined in Sect. 2 are endemic to the MapReduce model. Hadoop's implementation is strongly based around this model, and is therefore not suited to investigating alternative solutions.

Jumbo consists of two primary components, Jumbo DFS and Jumbo Jet.

3.1 Jumbo DFS

The Jumbo DFS is a distributed file system based on the Google File System. It uses a single name server to store the file system name space. Files are divided into large blocks, typically 128MB, which are stored on data servers. Each block is replicated to multiple servers, typically three, and the replicas are placed in a rack-aware manner for improved fault tolerance.

Jumbo's replica placement algorithm is currently similar to that of Hadoop. Some simple balancing is provided to prevent nodes from running out of disk space. Jumbo will also take into account how many clients are currently writing to a data server when choosing the location for a new replica, to prevent a single node getting swamped with too many simultaneous write requests. It is our expectation that we will refine the data placement algorithm in the future to provide for improved load balancing.

3.2 Jumbo Jet

The second component is Jumbo Jet, the data processing environment for Jumbo. It provides a programming model as well as an execution environment.

Jumbo Jet represents jobs as a sequence of stages, each of which performs a step in the data processing algorithm. Each stage is divided up into one or more tasks, where each task performs the same operation but on a different part of the data. Tasks provide parallelization and are the unit of scheduling for Jumbo Jet.

A stage reads data either from the DFS or from a channel connecting it to a preceding stage, and writes data to the DFS or a channel connected to a following stage. In order to divide input data from the DFS across multiple tasks, it is split into pieces, typically using DFS blocks as a unit. Intermediate data is partitioned using a user-specified partitioning function, so that each task in the stage receives the same partition from every task in the preceding stage.

Channels connect stages and represent the data flow between them. Channels control how data is partitioned, and how the pieces of each partition are merged. This means that unlike MapReduce we allow the user to decide what it wants to do with the intermediate data. In MapReduce, intermediate data is always sorted and grouped by key. In Jumbo Jet, you can choose to sort it (built-in functionality for this is provided), but in cases where it isn't needed, you don't have to.

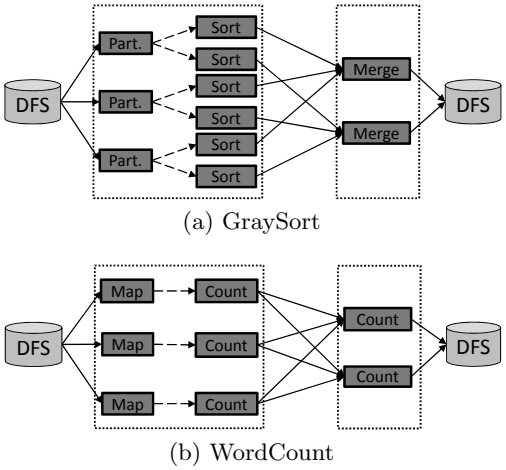


Figure 1: Examples of the structure of Jumbo Jet jobs.

Jumbo Jet provides several types of channels: file, TCP and in-process.

File channels are the most common type, and are similar to how intermediate data is treated in MapReduce; intermediate data is stored in files on the local disk, which are transferred over the network. This method offers maximum fault tolerance.

TCP channels provide a direct network communication channel between the tasks of two stages. This has the advantage of avoiding materializing the intermediate data, but it puts some constraints on the job structure and is not as fault tolerant.

Finally, in-process channels are used to provide a mechanism to combine multiple task types into a single, compound stage. Tasks that are connected using an in-process channel are treated as a unit by the scheduler and will be executed on the same node, in the same process. It is possible for an in-process channel to partition the data, in which case the outgoing file or TCP channel from the compound task will use that existing partitioning.

From a load balancing perspective, stages that read from the DFS behave in a similar manner as the map phase, while stages that read data from a channel behave in a similar manner as the reduce phase.

The structure of two simple jobs is provided in Fig. 1 to provide an example of Jumbo's processing model. Fig. 1(a) shows a sorting operation. It has two stages, the first of which is a compound stage using the in-process channel. This first stage partitions the data, and then sorts each partition. The second stage gathers the pieces of each partition and merges them. This mechanism is very similar to how sorting is accomplished in MapReduce.

Fig. 1(b) shows the Jumbo version of the word count sample that is often used to illustrate MapReduce. Although word count is a good example of how to use MapReduce conceptually, the counting method used – which sorts the words in the document – is actually very inefficient. A more straightforward method to do counting is to store the words in a hash table structure and update the counts incrementally as you scan the document. MapReduce does not allow you to use this approach.

The word count job in Jumbo has two stages. The first is a compound stage, the first part of which reads the words in the document and convert them to a (word, 1) key-value pair, the same as the approach used by MapReduce. The second part of the first stage inserts each pair into a hash table keyed by the word. The second stage of the job is identical to the second part of the first stage, this time using the counts produced by the first stage as input. This approach does not require that the intermediate data is sorted, and is significantly faster than the MapReduce method.

Job execution in Jumbo Jet is similar to Hadoop and Dryad. Job scheduling is handled by a single job server, which is responsible for assigning tasks to individual nodes in the cluster and recovering from task failures. Each node runs a task server which receives tasks to execute from the job server.

3.3 Complex Algorithms

In Sect. 2.3 we mentioned that MapReduce often requires the use of multiple jobs for complex algorithms, and that this can limit load balancing efficiency. One of the biggest advantages of Jumbo's data processing model is that we can easily represent these complex algorithms in a single job.

For example, the Parallel FP-Growth frequent item set mining algorithm proposed in [9] consists of three MapReduce jobs. The structure of this algorithm in MapReduce is shown in Fig. 2(a). It uses separate jobs for the parallel counting, PFP and aggregating phases of the algorithm, and also includes a grouping operation that is not done with MapReduce at all.

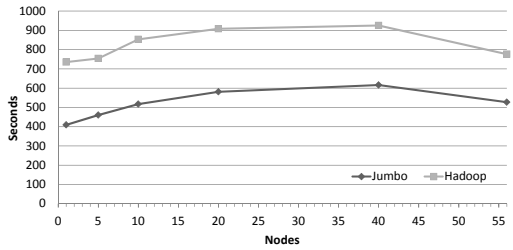
This means it suffers from the drawbacks mentioned in Sect. 2.3. For example, it is likely that the aggregation job would be able to do some of its work even with partial data. However, this is not possible because that job cannot start until the previous one has completed.

Jumbo's more flexible job structure allows us to represent the entire algorithm in a single job with six stages, as shown in Fig. 2(b). The first two stages perform the parallel counting operation; this can also benefit from the more efficient hash table counting approach mentioned earlier. The grouping operation is also part of the job structure, but is still not parallelized because this is not necessary. The last three stages perform the PFP and aggregation operations; here we can eliminate the map phase from the aggregation because it existed only to repartition the data, something Jumbo can do directly.

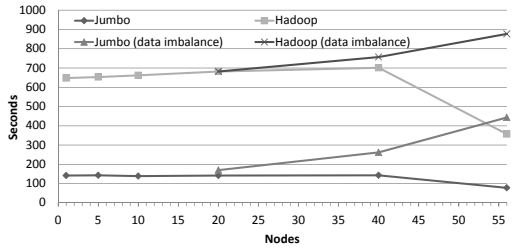
The scheduler will now be able to see the entire job structure at once, and will be able make better load balancing decisions. Although the PFP stages must wait until grouping is complete, Jumbo can still already schedule and start some of these tasks, and they can prefetch some data from the transaction database while they wait for the grouping data to be available. The aggregation phase can already begin its work even with partial data. These things would not be possible with Hadoop.

4. EXPERIMENTATION

We will demonstrate some of the load balancing issues discussed using two simple examples. We show two experiments on both Hadoop and Jumbo, using the GraySort and WordCount jobs, both of which are included as an example in Hadoop. The implementation for these jobs used in Jumbo was given in Sect. 3.1.



(a) GraySort



(b) WordCount

Figure 3: GraySort and WordCount performance for Hadoop and Jumbo. Up to 40 nodes, all nodes used are identical; only in the 56 nodes case was a heterogeneous environment used.

Two sets of nodes were used for this experiment: 40 older nodes, with 2 CPUs, 4GB RAM and one disk, and 16 newer nodes with 8 CPUs, 32GB RAM and two disks.

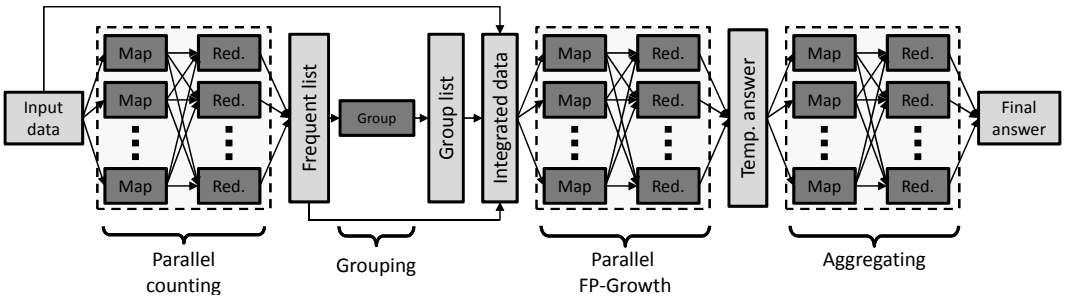
Fig. 3(a) shows the results the GraySort experiment. Up to 40 nodes, we used only the older nodes. Then for the 56 nodes experiment, the 16 newer nodes were added. We increased the amount of data as we increased the number of nodes, maintaining 4GB of data per node. This means that ideally, the results of each experiment should be the same, except for the 56 node experiment which should be faster.

In GraySort the size of the intermediate data is very large, and an external merge sort is required to sort it. Up until 20 nodes, the increasing number of merge passes required causes a slowdown in the sort time. From there to 40 nodes only a small slowdown is observed which is due to communication overhead. Both Hadoop and Jumbo show very similar profiles because they use a similar sorting method.

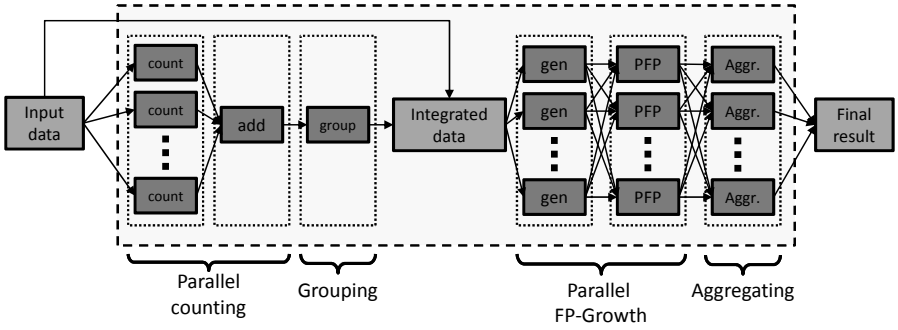
Jumbo's speed advantage is caused primarily by the way Hadoop transfers intermediate data between nodes; at one time many reduce workers may be reading data from the same node. On this particular cluster that has a very high impact on performance. Hadoop's sorting implementation is also more CPU intensive than Jumbo's.

When we go from 40 to 56 nodes, we expect to see a speed increase because the 16 additional nodes are much faster, and indeed this is the case. The self-balancing nature of tasks reading DFS data means that the map phase (or first stage in Jumbo) can finish sooner, increasing the speed of the job.

However, this speed increase is not optimal, because the reduce phase (merge stage in Jumbo) is affected by the issues mentioned in Sect. 2.2. The 16 faster nodes finish considerably sooner than the 40 slower nodes, as shown in Fig. 4. For



(a) MapReduce



(b) Jumbo

Figure 2: The Parallel FP-Growth algorithm with MapReduce and Jumbo. The MapReduce version uses three jobs, while the Jumbo version has only one.

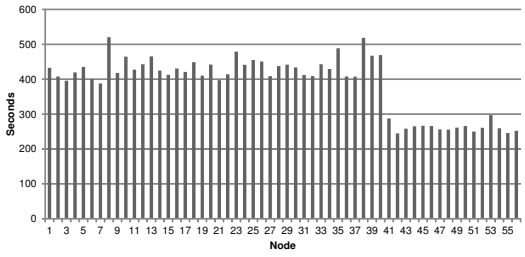


Figure 4: Execution times of individual nodes in the cluster for Jumbo GraySort. Nodes 1-40 are the older, slower nodes, while 41-56 are the faster ones.

the remainder of the job, the faster nodes are not utilized, wasting their processing power.

It can also be seen that even amongst the identical nodes, various external factors cause these nodes to also have varied execution times. It is therefore clear we need to be able to do dynamic balancing that can respond to these factors as needed.

The second experiment uses WordCount. Contrary to GraySort, this job has only very small amounts of intermediate data, and the reduce phase (second stage in Jumbo) has very little work to do and finishes just a few seconds after the last map task. Imbalance such as that demonstrated by GraySort will therefore not be seen.

Instead, this experiment will be used to demonstrate the effect of data placement. In this job we again increased the amount of data with the number of nodes, maintaining 2GB per node for each run, and first used our 40 slow nodes and then added the 16 faster ones. We then repeated the 20, 40 and 56 node experiments but this placed all the data for those jobs on only the first 10 nodes. This means that in the 56 node experiment the input data is still 112GB in size, but all of it is on the first 10 nodes and the remaining nodes need to read their input data from those nodes.

The result is shown in Fig. 3(b). With the data on all nodes, up to 40 nodes the result shows linear scalability, particularly for Jumbo. Jumbo is considerably faster than Hadoop because of the hash table counting method, which avoids the expensive sorting operation.

When going to 56 nodes we see a speed increase as expected. Hadoop shows a larger relative speed increase because WordCount in Hadoop is much more CPU intensive than in Jumbo, so it benefits more from the fast CPUs in the 16 new nodes. Because the reduce phase (second stage in Jumbo) has so little work, the imbalance observed in GraySort is not seen here.

However, if we put data on only 10 nodes, performance suffers considerably. The job's performance is now limited by the I/O speed of those 10 nodes, and the CPU speed of the remaining nodes can no longer be fully utilized. The more nodes we add, the bigger the imbalance, and the bigger the performance gap. For the 56 node experiments, the 16 faster nodes show only around 35% CPU usage for Hadoop, and around 10% for Jumbo (compared to 100% for both with the data properly distributed on all 56 nodes). This means a large amount of computing power is left unused.

It can also be seen that Jumbo has a bigger relative slowdown than Hadoop when the data is on 10 nodes. This is again because Hadoop's version of WordCount is much more CPU intensive. In fact, it's so CPU intensive that

when using 20 nodes Hadoop doesn't suffer at all from the data imbalance.

5. FUTURE WORK

For our future work, we intend to focus on the issue of complex algorithms such as Parallel FP-Growth. This issue encapsulates many of the other low-level issues, so we can address those within this context. This also offers the biggest opportunity to use the greater flexibility of Jumbo by focusing on algorithms that are not easily represented in MapReduce.

Absorbing heterogeneity in distributed systems has been studied in the past [11][6], but not much has been done for the kind of highly scalable, fault tolerant frameworks discussed in this paper. Some of the issues will be related, in which case we will attempt to apply existing solutions in this context.

But there are also new issues presented by this environment. The larger scale and flexible nature of the cloud means making assumptions about the environment is more difficult, and the data intensive nature of these systems means that the simple act of trying to move work to another node can cause additional delays because of the I/O incurred by the move. These issues must all be considered carefully.

Additional difficulty arises because in both MapReduce and Jumbo, tasks are arbitrary programs written in languages like Java or C# rather than a more structured approach like SQL. This makes it very hard to develop accurate cost estimates to base decisions on. Accurate progress indications are also difficult because of this, and these are vital in determining how and where to move work.

We will continue the development of Parallel FP-Growth and other complex data mining algorithms, and then build a load balancer that can address the issues described. We will modify Jumbo as needed to support our needs in this area, though it is our expectation that some of the techniques developed will also be applicable to other systems such as Hadoop.

Complementary to this issue is the problem of configuration. Determining the optimal configuration to use with Hadoop or Jumbo for any particular cluster is very hard. There are many parameters, and the effect of each of them is not always obvious. In heterogeneous environments, different nodes need different configurations, and the optimal configuration for one job might not work for another. As part of our efforts, we will attempt to let the system automatically determine the optimal value for some configuration parameters based on the environment.

Since scale is an important factor in all of this, we will also increase the scale of our experimental environment. A larger cluster for this purpose is already under construction.

6. CONCLUSION

Load balancing in heterogeneous clusters is a complex issue that is only made more complex by the large scale and operational complexity of modern distributed systems.

The development of highly scalable and fault tolerant systems such as MapReduce has enabled data processing with unprecedented scale. The issue of load balancing in these environments is however still largely unexplored. The simplistic load balancing abilities of existing frameworks, such as Hadoop's speculative execution, simply aren't sufficient to efficiently use all the resources in a heterogeneous cluster.

Going forward, the data processing capabilities of Jumbo will allow us to explore these issues and will give us the flexibility to deviate from the rigidity of MapReduce where necessary while still maintaining its properties of scalability and fault tolerance.

7. ACKNOWLEDGEMENTS

The author would like to thank Kazuo Goda for his help in preparing this paper.

8. REFERENCES

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

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

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

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[6] 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 *DEXA*, pages 182–192, 2002.

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

[8] 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, June 2007.

[9] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *RecSys '08*, pages 107–114, New York, NY, USA, 2008. ACM.

[10] 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 *SIGMOD '09*, pages 165–178, New York, NY, USA, 2009. ACM.

[11] M. Tamura and M. Kitsuregawa. Dynamic load balancing for parallel association rule mining on heterogenous pc cluster systems. In *VLDB '99*, pages 162–173, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *ICDE '10*, 2010.

[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. *OSDI*, 2008.