Optimizing MongoDB-Hadoop Performance with Record Grouping

by

MATTHEW ADAM JUSTICE

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

The University of Arizona

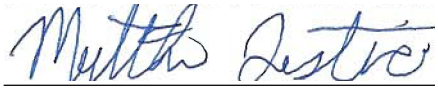
M A Y 2 0 1 2

Approved By:

John H. Hartman
Department of Computer Science

Statement by Author

This thesis has been submitted in partial fulfillment of requirements for a degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Signed: 

Abstract

Computational cloud computing is more important than ever. Since time is literally money on cloud platforms, performance is the primary focus of researchers and programmers alike. Although distributed computing platforms today do a fine job of optimizing most types of workflows, there are some types, specifically those which are not computation-oriented, that are left out. After introducing important players in the world of computational cloud computing, this paper explores a possible performance enhancement for these types of workflows by reducing the overhead that platform designers assumed was acceptable. The enhancement is tested in two environments: an actual distributed computing platform and an environment that simulates that platform. Along the way it becomes clear that computational cloud computing is far from perfect and its use can often deliver surprising results. Regardless, the presented solution remains viable and is capable of increasing the performance of particular types of jobs by up to twenty percent.

Table of Contents

1. Introduction

The next big problem in computer science involves the processing and analysis of large amounts of data. Whether it is Google indexing millions of pages, financial institutions processing transactions, or research institutions making the next breakthrough in science, our ability to process large amounts of data is more important than ever. This paper is part of a larger project to help plant scientists organize and analyze large amounts of genome data in an effort to better understand world agriculture. After giving a primer in the fields of cloud computing, NoSQL databases, and distributed computing, the ever-growing issue of performance will be discussed. This paper will then focus upon optimizing cloud infrastructure through a process called "record grouping".

"Cloud computing" is a bit of a buzzword these days. The term originates from how the Internet is normally depicted on network diagrams, as a cloud. The concept is to turn computing into a utility, much like the electricity grid, but over the Internet. When a client has a job to perform, he/she determines what type of cloud resources he/she wants (normally in terms of CPU cycles and performance-critical specs like CPU speed and memory) and for how long he/she want them for (this process is normally called "provisioning"). After deploying and running the job, the client collects the results and gets to "leave" – most cloud providers do not require contracts or long-term commitments. The beauty of this, is that companies no longer need to invest in expensive hardware, complex infrastructure, and high-salaried IT professionals.

As one may imagine, cloud computing has had huge implications for individuals and organizations in nearly every industry. Financial institutions love it because they can have their nightly transactions processed for pennies on the dollar. Universities and other research organizations often need to crunch large amounts of data but don't have the funding or knowledge to build their own group of supercomputers. Even end

users can see cloud computing in action when they are using services like Google's Gmail and Apple's iCloud. When technical users all share resources and let a few companies take care of the development and maintenance of large clusters of high-end machines, everyone wins.

Cloud computing is an incredibly broad concept that covers many different applications, platforms, and use cases. In this paper, I plan on focusing on how research organizations use cloud services. Researchers present an interesting use case in multiple ways. First, many researchers produce a large amount of data on a daily basis but have no means to handle and analyze the data in an effective way. Second, researchers usually have the goal of sharing data with others, which makes cloud storage (think Dropbox and Google Docs) an excellent choice. Finally, researchers often have to carefully balance two critical factors: time and money. In fact, the project this thesis falls under closely examines that problem: how researchers choose the cloud solution that best fits their budget and time constraints.

1.1. Problem Overview

Performance is always the name of the game. How can the input parameters or environment be modified to make the job run faster? It turns out that one of the problems of cloud computing is that there is an expectation that computations to be difficult and time-consuming. After all, if the computations are simple, it might just be more efficient to do them on a consumer-grade computer. The problem is that these computations are not very complicated – there are just too many to do. The distributed computing platform I'm using works against researchers when their computations are short because the platform spends a significant amount of time performing set-up for the tasks and distributing the data. One could simply modify the various configuration options that control our environment in an attempt to get the platform to be friendlier to our job. However, this is insufficient and doesn't produce significant results. Distributed computing platforms are just too computation

oriented for this purpose.

"Record grouping" will allow us to feed more data records to our distributed computing platform than a usual configuration. Instead of feeding our processing function one record at a time, we'll feed it a record that contains multiple records inside of it, in array form. Now, each of the computation tasks will have more to do, offsetting the overhead of task setup and data distribution. The specifics of record grouping will be introduced later after MongoDB and Hadoop have been introduced.

Using record grouping, I will be able to improve the performance of a MongoDB-Hadoop job by up to 20%. Unfortunately, I will encounter issues with Hadoop and its initialization process that brings to question whether Hadoop is even suitable for this task at all. Regardless, I present record grouping as a relatively easy means by which to improve your workflow performance.

2. MongoDB

The first problem to solve in processing large amounts of data is storage. Often the data are stored in flat text files. For computational purposes, it cannot kept in this form. There is too much OS overhead from reading data from files constantly and there is no effective way of sorting and grouping the data. A first thought may be to use the traditional relational database management systems that computer scientists have grown to love. Systems created by Microsoft, Oracle, and IBM are great for managing complex data designs that require consistency among data (for example, a purchase order is associated with a client). However, these systems begin to show great weakness when asked to work with more than a few million rows. For this, one should turn to non-relational systems, called "NoSQL".

These systems ditch the idea of creating intentional database design and forced data relationships for a schema-less system that focuses on performance and scalability [2]. I have selected MongoDB for a few key reasons: 1) it is free and open-source; 2) there are multiple APIs available in numerous programming languages; and 3) it integrates nicely into Hadoop, a system to be discussed later.

MongoDB has some different terminology to introduce. Instead of tables, there are "collections". Instead of rows/records, there are documents (in a format called BSON). Consider the following document that might be in a university's student database:

```
 { "sid":  12345,
"name":  "Matt Justice",
"department":  "Computer Science",
"email":  {
           "personal":  "...",
           "school":  "..."
         }
"active":  true
}
```

Note the lack of data type definitions and how it is possible to nest documents inside other documents. Also note how the entire structure of the document is defined here. This is what allows MongoDB to be "schema-free". Each document can have a unique schema from other documents, and the schema can be changed at any time. This makes for great flexibility with NoSQL systems (although we lose the consistency of traditional RDBMs). Queries are defined in the same format (therefore, queries themselves are documents). For example, if one wanted to find all the Computer Science students in this collection, one might issue a query like:

```
{"department":  "Computer Science"}
```

This would match all records that have the `department` field equal to "Computer Science". If a document doesn't have that field, it is ignored.

Surprisingly, MongoDB actually supports a wide array of query operators and constructs, allowing one to replicate a large number of traditional SQL queries as

MongoDB queries. I won't be detailing MongoDB's query language in this paper. However, one should observe here that MongoDB's document structure allows a user to split data among multiple MongoDB machines, in a process called "sharding". The ability to split the data and distribute it to multiple machines becomes extremely important as I begin to discuss MapReduce and its uses in processing large amounts of data.

3. The MapReduce Paradigm

In 2004, Google released a now relatively well-known paper on a programming model they called "MapReduce". The model creates a means for programmers to easily write applications that run in a distributed manner on a large number of machines (called a *cluster*). Obviously, distributed computing is not a new concept, but Google's model abstracts out all complicated details and logistics that go along with parallel processing (scheduling tasks, distributing data, and handling machine failures) [3]. MapReduce defines two primary operators on the dataset: `map` and `reduce`. Map takes in a key/value pair and outputs an intermediate key/value pair. There is no limit on the number of intermediate pairs one can output. The keys become important in the reduce step, in which the intermediate data pairs are grouped together by the key and a final key/value pair are produced. MapReduce is extremely powerful because it can be used on any type of data in any format that can be effectively split among nodes. However, it is a bit too abstract to understand without a concrete example. Here is a quick overview of the classic MapReduce example, called Word Count.

3.1. Word Count Example

Word Count is a classic computer science problem, usually for first-year programming students. Given a text file, the task is to count the occurrences of different words and output a list in the form of `(<Word>, <Count>)`. It is easy to imagine how this task is accomplished using traditional programming techniques – scan through the file and every time a new word is encountered, add it to an array and set its count

to 1. Whenever that same word is encountered again, increment that word's count. However, this task is the perfect example for how to easily parallelize operations. It is possible to process each line of a text file completely independently from other lines, and then combine the data in a central location and print out the results. So, how exactly will this work? Here is the workflow:

1. Input text files are split by line.
2. The controlling node assigns a block of input lines to each computing node and sends it the data.
3. The compute node runs `map` on the input line, producing 1 intermediate data pair for every word. In this example, the pair `(<word>, 1)` is always emitted.
4. After running through the input, a compute node sends its intermediate data pairs to the node designated to perform the `reduce` operation.
5. The `reduce` operation is run once for each unique key (or word, in this case). In Word Count, the values of the intermediate pairs are simply summed up. Since all of our intermediate pairs have a value of 1, the result is an occurrence count for a particular word.
6. The controlling node receives the results and outputs the list to the console or output text file.

Here's what this might look like in pseudo-code:

```
function MAP(line)
    while line has words do
        emitIntermediatePair(currentWord, 1)
    end while
end function
function REDUCE(key, List< values >)
    sum ← 0
    for all values do
        sum ← sum + value
    end for
    emitOutputPair(key, sum)
end function
```

The specific source code for thee Word Count example is not given here as it is readily available on the Internet in multiple languages.

3.2. Distribution and Reliability

When distributing tasks, Google recommends using a "rack-aware" file system. The idea is that each node should know which rack of servers it is on and its location in that rack. Then, when a task is scheduled, MapReduce should take into account the location of the data when selecting compute nodes. The goal is to reduce the amount of bandwidth used in transferring data between machines within the cluster. Small clusters rarely implement this file system because it isn't worth it, but one can imagine a large data center benefiting from this greatly.

Compute nodes are expected to send a "heartbeat" to the controlling node with a status update so that the controller knows if a node goes down. If a node goes down, the controller is expected to re-assign the tasks to another node. With this requirement, the MapReduce paradigm expects a cluster to be relatively large so that a single or a few nodes going dead will not stop the task from completing. If the task is a particularly long one, the compute node is expected to send its results at a given time interval – preventing one from losing the data it has already produced.

3.3. Language Support

As a paradigm, MapReduce is language-agnostic. There are MapReduce implementations written for nearly every programming language one can imagine, including Java, C, C#, Perl, Python, Ruby, and PHP. MapReduce algorithms are most commonly written in Java because of Hadoop, the most popular MapReduce platform.

3.4. Other Notes

It should be noted that MapReduce does define other operations that occur between the Map and Reduce steps, normally involving the shuffling of data between compute nodes. I won't be discussing these operations here as they aren't relevant to our task-at-hand.

4. Hadoop

Apache Hadoop is the "de-facto" standard for MapReduce. It came about in the early 2000s as an effort to build a search engine from scratch. It wasn't until Google's now infamous paper in 2004 on MapReduce did the project receive its real jumpstart. Around 2006, Yahoo! hired Hadoop's creator and gave him his own team [4]. Yahoo! was able to improve Hadoop enough that it used Hadoop to produce its public-facing search engine index. In 2008, the project moved to Apache and became a community project. It is now used by some of the largest technology companies, including Facebook, Google, and IBM [5]. It is for this reason that Hadoop was selected.

4.1. Node Types

There are three main parts of a Hadoop cluster:

JobClient Responsible for submitting the job to the Hadoop cluster.

JobTracker Receives submitted jobs from JobClients, schedules them (using a FIFO queue), and sends out the necessary data.

TaskTracker Runs the tasks sent to it by the JobTracker. This node is actually responsible for performing the `map` and `reduce` operations.

4.2. Job Execution

The process of executing a Hadoop job is very simple from the high-level point of view, and quite complicated from the low-level point of view. I will avoid the specific details and give a brief overview. First, the JobTracker splits the input data into configuration-specified blocks (normally called the "split size"). Each split then turns into a task, which is run by a TaskTracker in a separate Java Virtual Machine (JVM). After completing work on a specific split, the TaskTracker sends the data to the node performing the `reduce` operation and moves onto the next split.

4.3. Configuration Options

One of Hadoop's key features is its large number of configuration options. This allows Hadoop to adapt to various types of jobs and clusters. Each job has over 100 configuration options, most of which are extremely advanced and simply inherited from the system default. I will examine a few important configuration options that I'll set in our experiment later.

- Each node has a number of "slots", which is the number of tasks that can run on that node at once (normally defaults to 10).
- The split size specifies the size of a split, the amount of data processed in a task.
- The "Task JVM Reuse" switch allows the TaskTracker to reuse the Java Virtual Machine for numerous tasks instead of launching a new one for each task.
- Finally, one can send standard JVM configuration options to control the virtual machines launched to perform tasks.

Now that the various technologies used in the experiment have been introduced, I can begin discussing the type of data being used and how the experiment was performed.

5. SNPS Data & Queries

The project this paper falls under is in cooperation with iPlant Collaborative, a University of Arizona organization working to bring computing infrastructure to the world of plant sciences.[1] For plant scientists, DNA sequences hold the key to creating stronger and more productive crops. Most genome sequencers produce nearly a terabyte of data each day – a lot to crunch through. To best simulate the environment by which these scientists need to crunch data, this experiment uses actual DNA sequencing data, called SNPs. I'll focus on how the data is formatted and what types of queries scientists wish to perform on this data.

5.1. Data Format

SNP data contains numerous fields that help scientists identify and understand the sequencing data. The majority of the data is not relevant for our purposes, with the exception of one field: the "RS" number. This unique number is assigned by a central database and used by scientists to effectively communicate about the sequence. This field is used as the primary key. The rest of the fields contain the expected DNA data (an allele) – each field is one of the famous nucleotides: A, T, C, or G. Another possible value is a dash, which indicates that the specific nucleotide is unknown.

5.2. Query Types

Plant scientists are most interested in doing comparisons in their queries. Often, the queries ask for two different records and want to see which alleles are different. Similarly, a scientist may wish to select two alleles and compare those two fields for each of the records. Both of these queries can be done with a solid knowledge in a SQL environment. The queries I am interested in are aggregate queries – in other words, do something for each record and combine that information with the results from other rows. In addition to fitting well with the MapReduce paradigm, these queries would

take an enormous amount of time when run in a linear fashion, especially on the average workstation machine. With MapReduce and cloud resources, these queries can be done in a reasonable amount of time for a reasonable cost.

A third type of query is often requested by scientists – the temporal (time-based) query. Scientists are often conducting long-term research to see how the genetic makeup of their sample group changes over time. Knowing how a certain subset of the sample is performing versus other subsets is extremely important. Unfortunately, MapReduce is not an ideal solution for temporal queries because it is more comparison-based than aggregate-based.

6. MongoDB-Hadoop Adapter

I should mention briefly the piece of software used to connect MongoDB and Hadoop. Without modification, Hadoop only reads text files. If one wishes to allow other sources of data for Hadoop, there are various Java classes that must be written to translate Hadoop's requests for data to instructions for the source data platform. This adapter is maintained by the company that created MongoDB, but it was still in beta when used for this experiment. It is not necessary to know a lot about the adapter besides the fact that the adapter always feeds the `map` function a single JSON from the source collection.

7. Experimentation

7.1. Grouping Records

"Record grouping" is actually quite simple. Instead of taking each data row from a text file and reading it as its own MongoDB document, it is read in as a "sub-document". The parent document will therefore have an array of these sub-documents. Therefore, during our MapReduce operations, the array is iterated through, treating each sub-document as if it was its own independent document. This allows for the creation of fewer splits of the collection, which produces fewer map tasks to perform. To create collections that use record grouping, we need an importer. MongoDB

comes with its own import utility for reading from files, but we must create our own to handle the grouping of records. *Appendix B* contains the code for two importers: `MongoGroupImporter` and `MongoRecordGrouper`. They both perform the same task, with one exception – GroupImporter reads its data from a text file, while RecordGrouper reads its data from an existing ungrouped collection.

7.2. The Query

To prove that record grouping is an effective way to increase performance, I need to devise a realistic MapReduce query and run it on the same data set with different sized record groupings (recall that a document is how MongoDB sees information – the term *record* here refers to a single RS number). For this query, I modify the classic Word Count example presented in the Hadoop section so that given a certain nucleotide, the query should group the records by how many times that nucleotide is present. I call this query "Frequency Count", and the code for this query is made available in *Appendix B*. I will always use 'N' as the character to match – this selection is arbitrary, but it must be kept constant. The specifications of the systems being used in this experiment are available in *Appendix A*. The single variable will be "RSize", which is defined as the number of records per document. I have selected 16 of these values based on initial experimentation.

7.3. Cluster Settings

In the Hadoop section, I mentioned a few important configuration options and alluded to the fact that I will need to set them. Some initial benchmarking showed that for the cluster, the split size should be set to 16MB. This setting is a reflection of the cluster configuration and not the specific query. In addition, I let the Hadoop job use the default number of "slots" for the cluster, which is 10. I do not enable task reuse because the goal is to reduce the number of tasks created (so the overhead is reduced). Reusing the JVM does not accomplish this. Each Hadoop job is run independently

– in other words, each job run will get the 4-node cluster to themselves. I ran each RSize 5 times, and then graphed the averages.

7.4. Cluster Results

The graph of our results is presented below. The X-axis is the RSize, and the Y-axis is the average time in minutes.

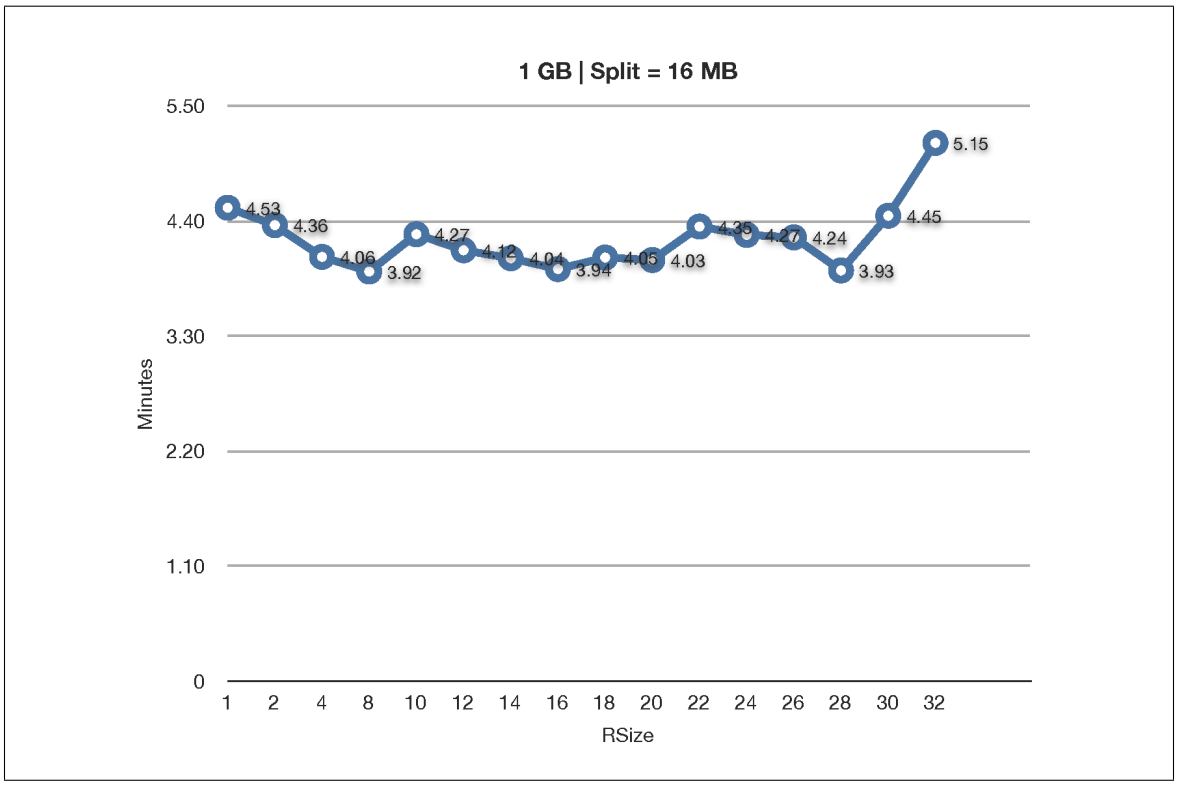


Figure 7.1: Results of running FrequencyCount on Hadoop cluster

It is clear that this graph is inconclusive and does not demonstrate a general trend. So what went wrong? I suspect that Hadoop is the problem, given the amount of time taken to run the query. Given the specifications of the machines we are using, and assuming the network connections are using Gigabit Ethernet, this job should run in less time.

To test this hypothesis, we add a command line option to our query and call it "short-circuit mode". The idea is to test the amount of time it simply takes to

initialize a Hadoop job and distribute the various pieces of data. When `map` gets called in this mode, the function simply returns. It does not perform any data processing or emit any intermediate pairs. Since no pairs are emitted, the `reduce` step won't run. If this time comes reasonably close to the times observed in the first experiment, one can conclude that Hadoop is the culprit.

This hypothesis turns out to be true – the average time to run the query in short-circuit mode with RSize = 1 is 4.03m, compared to 4.53m in normal mode. It is clear then that the vast majority of processing time is spent simply using Hadoop and not actually processing any data. Hadoop simply includes too many variables for users to run this experiment on it and expect significant results. To prove that record grouping is still effective, I will need a Hadoop simulator.

7.5. Simulator Settings

In simulating Hadoop, one must be careful about which features and included and excluded. The simulator should run on a single machine (the one that has the data on it). Also, the simulator should be able to perform splits and distribute the work locally. Finally, the simulator should do its best to mirror how tasks are actually launched by isolating each task in their own JVM and limiting the number of tasks running at a time.

I have written such a simulator and have included a link to the JAR file in *Appendix B*. For simplicity and consistency, the simulator uses a constant value of 10 maximum tasks at a time and a split size of 16MB. Although the framework by which the query will run is different, the query code and other variables remain largely untouched – RSize should be our only variant. I will also increase the number of runs of each RSize to 10 instead of 5 so that I can gather a more accurate average.

7.6. Simulator Results

I now present the graph for when FrequencyCount is run on our simulator. The X-axis is still the RSize, but the Y-axis is now in seconds. Also, the method of

calculating the averages changes slightly. The first run of every RSize seems to run abnormally along (at least double the average of other runs). This is likely a caching issue – after the first run, the operating system places the MongoDB collection in its cache, making the first run and access very expensive, and subsequent accesses much faster. So, to ensure that this does not affect the averages, I drop the minimum and maximum result for each RSize and average the rest. The timings were performed by the Linux command `time`.

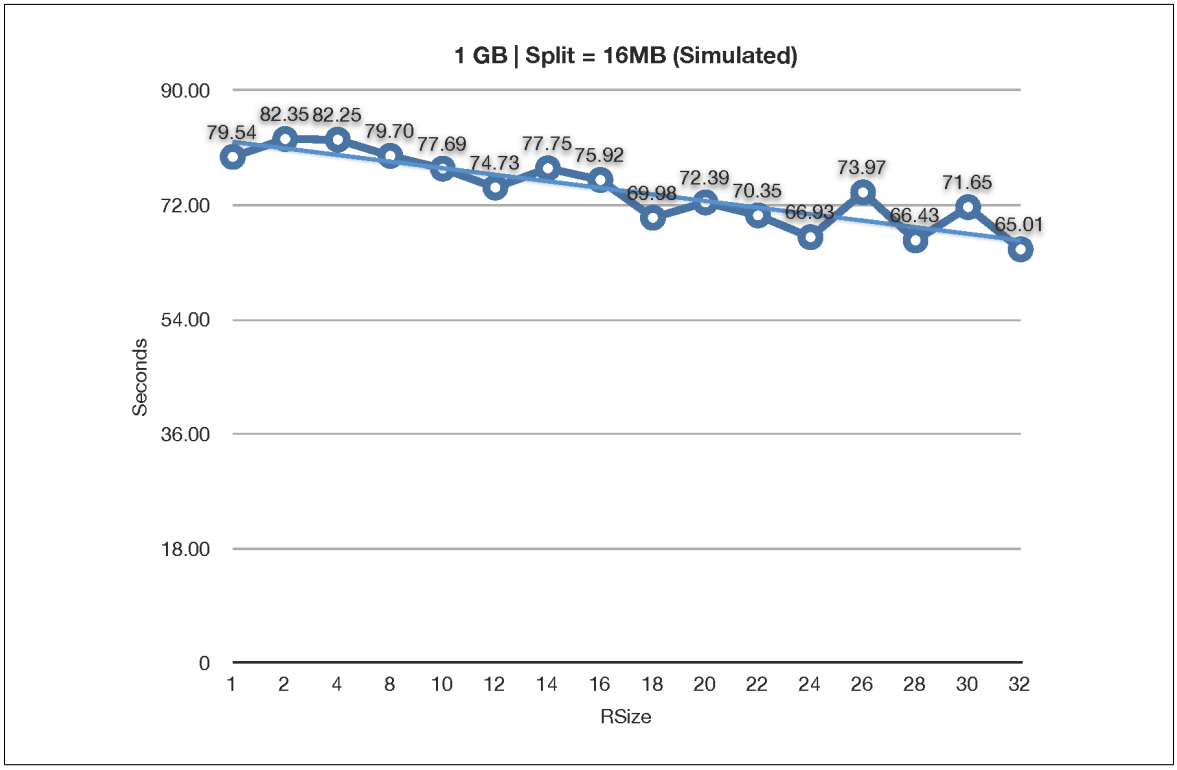


Figure 7.2: Results of running FrequencyCount on Hadoop simulator

This graph presents a much better (yet not perfect) representation of what is occurring. Besides a few anomalies, the trend is generally down, as shown by the linear trend-line drawn. At RSize = 1, the time was nearly 80 seconds. At RSize = 32, the time drops to 65 seconds – an improvement of nearly 20%.

The graph's initial increase can be explained by the overhead in the process of record grouping. When grouped documents are stored, each record is contained as

an entry in an array. To represent the array, the MongoDB Java driver converts the JSON array into a Java list structure. Once there are enough records per document to overcome this overhead, the times begin decreasing. I can attribute the other spikes to the way splits are divided. Rarely will the RSize allow one to split the collection into perfectly even splits – one of the splits is bound to end up with different number of documents in it. Ideally, one would like this abnormal split to be reasonably close in size to the regular number of documents per split. Minimally, one would like the number of documents in the abnormal split to be high so that JVM is not launched to perform a very small amount of work. At particular values of RSize, the size of the abnormal split becomes very small (less than 20). In these scenarios, the computing platform does not get the opportunity to overcome the overhead of launching a JVM, leading to longer processing times.

8. Conclusion

Through the process of creating this paper, I have observed some very interesting behavior. Although MapReduce and Hadoop are powerful platforms for distributed computing, they are far from perfect. The amount of time necessary to initialize the environment can be overwhelming and unpredictable. Even when one can isolate the performance optimization from the Hadoop environment, we find other factors that hinder our performance gain. Record grouping seems to be a viable method to improve performance, but it is not without its issues. The time necessary to import the data in the grouped format can ruin an efficient workflow. Grouped records are difficult to deal with if using the data outside of the workflow (like another database application). Regardless, I have presented a solution to the problem proposed. Not everyone is using computational cloud computing for the same types of workflows. Therefore, it imperative that the computer science community explore ways to improve performance for all types of workflows, including those which require simple calculations.

References

[1] iPlant Collaborative, 2012. [Online; accessed 27-April-2012].

[2] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, pages 483 –488, 29 2011-sept. 2 2011.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[4] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[5] Wikipedia. Apache Hadoop — Wikipedia, The Free Encyclopedia, 2012. [Online; accessed 16-April-2012].

A. Experimental Setup

MongoDB

The MongoDB server is a 2.66GHz dual six-core machine (12 physical cores) with 72GB of RAM and a 1TB hard drive. It is running MongoDB version 2.0.4.

Hadoop Cluster

The Hadoop cluster has four nodes with Hadoop version 0.20. Each node is a 2.66GHz dual quad-core machine (8 physical cores per node) with 24GB of RAM and 1TB disks.

MongoDB-Hadoop Adapter

The adapter that allows the feeding of input from a MongoDB database to a Hadoop cluster was in late beta stages when used in this experiment. As of publication, the adapter is at its first release candidate and is publicly available at `https://github.com/mongodb/mongo-hadoop`.

Networking

The MongoDB server and Hadoop cluster are on the same networking switch. However, that switch is used for other computing clusters as well, so it is not dedicated to the Hadoop cluster. To account for network traffic, our experiments are run at off-peak hours, either after 5pm on weekdays or on weekends.

B. Java Code

You can find the full applications referenced in this paper in runnable JAR format at `http://www.cs.arizona.edu/people/mjustice/honors-thesis/`.

Listing B.1: FrequencyCount.java

```
package hadoop;
import java.io.IOException;
import java.util.List;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.bson.BSONObject;
import com.mongodb.hadoop.MongoInputFormat;
import com.mongodb.hadoop.MongoOutputFormat;
import com.mongodb.hadoop.util.MongoConfigUtil;
public class FrequencyCount {
  @SuppressWarnings("unused")
  private static final Log log = LogFactory.getLog(FrequencyCount.
      class);
  public static class FCMapper extends
      Mapper<Object, BSONObject, IntWritable, Text> {
    @SuppressWarnings("unchecked")
    public void map(Object key, BSONObject value, Context context)
        throws IOException, InterruptedException {
      // To determine Hadoop overhead, we can short-circuit this so
          nothing happens
      if( context.getConfiguration().getBoolean("shortCircuitMode",
          false) == true) return;
      // Figure out if we are in group mode or not
      if( context.getConfiguration().getBoolean("groupMode", false)
          == true ){
        List<BSONObject> mainArray = (List<BSONObject>) value.get("
            mainArray");
        // Loop through all the records in the array
        for (BSONObject value2 : mainArray) {
          mapHelper(value2, context);
        }
      }else{
         mapHelper(value, context);
      }
    }
```

```
  private void mapHelper(BSONObject value2, Context context) throws
      IOException, InterruptedException{
    // Initialize variables
    String rsNumber = value2.get("rs#").toString();
    int currentCount = 0;
    // We're ignoring the key, but we'll break down the BSON object
    // looking for certain types of field names
    for (String fieldName : value2.keySet()) {
      if (fieldName.charAt(0) == 'Z'
          || fieldName.charAt(0) == 'M') {
        String currValue = value2.get(fieldName).toString();
        if (currValue.charAt(0) == context.getConfiguration()
            .get("matchCharacter").charAt(0)) {
          currentCount++;
        }
      }
    }
    // We've iterated through all the fields, now just set what we
    // need to set
    context
        .write(new IntWritable(currentCount),
            new Text(rsNumber));
  }
}

public static class FCReducer extends
    Reducer<IntWritable, Text, IntWritable, Text> {
  public void reduce(IntWritable key, Iterable<Text> values,
      Context context) throws IOException, InterruptedException {
    StringBuilder sb = new StringBuilder();
    for (Text t : values) {
      sb.append(t.toString() + " ");
    }
    context.write(key, new Text(sb.toString().trim()));
  }
}
public static void main(String[] args) throws Exception {
  final Configuration conf = new Configuration();
  String collectionName = args[0];
  conf.set("matchCharacter", args[1]);
  conf.setInt("mongo.input.split_size", Integer.parseInt(args[2]));
  conf.setBoolean("groupMode", Boolean.parseBoolean(args[3]));
  conf.setBoolean("shortCircuitMode", Boolean.parseBoolean(args[4])
      );
  MongoConfigUtil.setInputURI(conf,
      "mongodb://lecter.iplantcollaborative.org/gnomes."
          + collectionName);
  MongoConfigUtil.setOutputURI(conf,
      "mongodb://lecter.iplantcollaborative.org/gnomes."
```

```
            + collectionName + "_out");
    System.out.println("Conf: " + conf);
    final Job job = new Job(conf, "frequency count");
    job.setJarByClass(FrequencyCount.class);
    job.setMapperClass(FCMapper.class);
    job.setCombinerClass(FCReducer.class);
    job.setReducerClass(FCReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(Text.class);
    job.setInputFormatClass(MongoInputFormat.class);
    job.setOutputFormatClass(MongoOutputFormat.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Listing B.2: StringArrayWritable.java

```
package hadoop;
import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
public class StringArrayWritable extends ArrayWritable {
  public StringArrayWritable() {
    super(Text.class);
  }

  public StringArrayWritable(Writable[] values){
    super(Text.class, values);
  }

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof StringArrayWritable) {
      StringArrayWritable comparison = (StringArrayWritable) obj;

      if (comparison.get().length == get().length) {
        Writable[] thisWritables = get();
        Writable[] comparisonWritables = get();

        for (int i = 0; i < comparisonWritables.length; i++) {
          if (!comparisonWritables[i].equals(thisWritables[i])) {
            return false;
          }
        }

        return true;
      }
    }
```

```
    return false;
  }

  @Override
  public String toString() {
    if (get().length != 0) {
      StringBuilder builder = new StringBuilder();

      for (int i = 0; i < get().length; i++) {
        builder.append(get()[i]).append(", ");
      }

      return builder.toString();
    } else {
      return "";
    }
  }
}
```

Listing B.3: MongoGroupImporter.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashMap;
import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.Mongo;
import com.mongodb.MongoException;
public class MongoGroupImporter {
  private HashMap<String, String> arguments = new HashMap<String,
      String>();
  private String[] fieldNames;
  private DBCollection currColl;
  private String splitString;
  public static void main(String[] args) {
    new MongoGroupImporter(args);
  }
  public MongoGroupImporter(String[] args) {
    // Loop through all the command line arguments and load up the
       HashMap
    processOpts(args);
    // Test required arguments
    if (!arguments.containsKey("host") || !arguments.containsKey("db"
       )
```

```
    || !arguments.containsKey("coll")
    || !arguments.containsKey("file")
    || !arguments.containsKey("type")) {
  System.out.println("Usage: MongoGroupImporter");
  System.out
      .println("--host: The MongoDB host to connect to (required)
         .");
  System.out.println("--db: The database to use (required).");
  System.out.println("--coll: The collection to use (required).")
     ;
  System.out.println("--file: The file to import (required).");
  System.out
      .println("--type: The type of file to import.  Options: csv
         ,tsv (required).");
  System.out
      .println("--rsize: The number of rows to group into a
         single record. Defaults to 1");
  System.out
      .println("--overwrite: Clear the collection before
         importing.");
  System.exit(1);
}
// Set the split string
if (arguments.get("type").equals("tsv")) {
  splitString = "\\t";
} else if (arguments.get("type").equals("csv")) {
  splitString = ",";
} else {
  System.out.println("Unknown file type given.");
  System.exit(1);
}
// Try to create a MongoDB connection
Mongo connection = null;
try {
  connection = new Mongo(arguments.get("host"));
} catch (UnknownHostException e) {
  System.out.println("Could not connect to host.");
  System.exit(1);
} catch (MongoException e) {
  e.printStackTrace();
  System.exit(1);
}
// Grab the DB specified
DB currDB = connection.getDB(arguments.get("db"));
// Get the collection specified
currColl = currDB.getCollection(arguments.get("coll"));
// Clear the collection if we are supposed to
if (arguments.containsKey("overwrite")) {
  currColl.remove(new BasicDBObject());
```

```
    }
    // Store the size of the collection before we start
    long collectionSize = currColl.count();
    // Now we are ready to process the file, so send it off...
    int linesRead = processFile();
    // Tell the user all the exciting stuff we did
    System.out.printf("Read %d data lines\nInserted %d records\n",
        linesRead, currColl.count() - collectionSize);
    // Now close the connection
    connection.close();
  }
  private int processFile() {
    // Open the file for reading
    BufferedReader fileReader = null;
    try {
      fileReader = new BufferedReader(new FileReader(arguments
          .get("file")));
    } catch (FileNotFoundException e) {
      System.out.println("File specified not found.");
      System.exit(1);
    }
    // Grab the first line and send it off to set field names
    try {
      if (fileReader.ready()) {
        fieldNames = fileReader.readLine().split(splitString);
      } else {
        System.out.println("File is empty -- exiting.");
        System.exit(1);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    // Figure out what our recordSize will be
    int recordSize = (arguments.containsKey("rsize")) ? Integer
        .parseInt(arguments.get("rsize")) : 1;
    ArrayList<BasicDBObject> records = new ArrayList<BasicDBObject>(
        recordSize);
    // Keep track of how many records and lines we have inserted
    int linesRead = 0;
    // Now we iterate through the file
    try {
      while (fileReader.ready()) {
        records.add(processLine(fileReader.readLine()));
        linesRead++;
        // If we now have recordSize records, then write it out
        if (records.size() == recordSize) {
          currColl.insert(new BasicDBObject().append("mainArray",
              records));
          records.clear();
```

```
      }
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
  // If we have something left at the end, insert it as well
  if (!records.isEmpty()) {
    currColl.insert(new BasicDBObject().append("mainArray", records
        ));
    records.clear();
  }
  // Close the file reader, like a good programmer
  try {
    fileReader.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
  // Return number of data lines we read
  return linesRead;
}
private BasicDBObject processLine(String currentLine) {
  BasicDBObject toReturn = new BasicDBObject();
  // Split up up the current line
  String[] tokens = currentLine.split(splitString);
  // Loop through all the tokens, adding to our DBObject
  for (int i = 0; i < tokens.length; i++) {
    toReturn.append(fieldNames[i], tokens[i]);
  }
  // Return the result
  return toReturn;
}
private void processOpts(String[] args) {
  int i = 0;
  while (i < args.length) {
    if (args[i].substring(0, 2).equals("--")) {
      if ((i + 1) < args.length) {
        if (args[i+1].length() > 2 && args[i + 1].substring(0, 2).
            equals("--")) {
          // Both this argument and the next one are standard
          // arguments
          // So we just store this one and move on
          arguments.put(args[i].substring(2), null);
          i++;
        } else {
          // The next argument goes along with this argument
          arguments.put(args[i].substring(2), args[i + 1]);
          i += 2;
        }
      } else {
```

```
            // This must be the last argument, so just store it if we
            // get here
            arguments.put(args[i].substring(2), null);
            i++;
          }
        } else {
          // Arguments that don't begin with a dash and aren't
              associated
          // with another argument
          // should be ignore
          i++;
        }
      }
    }
  }
```

Listing B.4: MongoRecordGrouper.java

```
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.HashMap;
import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.Mongo;
import com.mongodb.MongoException;
public class MongoRecordGrouper {
  private HashMap<String, String> arguments = new HashMap<String,
      String>();
  private DBCollection sourceCollection;
  private DBCollection destCollection;

  public static void main(String[] args){
    new MongoRecordGrouper(args);
  }

  public MongoRecordGrouper(String[] args){
    // Loop through all the command line arguments and load up the
        HashMap
    processOpts(args);
    // Test required arguments
    if (!arguments.containsKey("host") || !arguments.containsKey("db"
        )
        || !arguments.containsKey("source")
        || !arguments.containsKey("dest")
        || !arguments.containsKey("rsize")) {
      System.out.println("Usage: MongoRecordGrouper");
```

```
      System.out
          .println("--host: The MongoDB host to connect to (required)
              .");
      System.out.println("--db: The database to use (required).");
      System.out.println("--source: The source collection to use (
          required).");
      System.out.println("--dest: The collection used for output (
          required).");
      System.out
          .println("--rsize: The number of rows to group into a
              single record (required).");
      System.out
          .println("--overwrite: Empty the destination collection
              before importing.");
      System.exit(1);
    }
    // Try to create a MongoDB connection
    Mongo connection = null;
    try {
      connection = new Mongo(arguments.get("host"));
    } catch (UnknownHostException e) {
      System.out.println("Could not connect to host.");
      System.exit(1);
    } catch (MongoException e) {
      e.printStackTrace();
      System.exit(1);
    }
    // Grab the DB specified
    DB currDB = connection.getDB(arguments.get("db"));
    // Get the collections
    sourceCollection = currDB.getCollection(arguments.get("source"));
    destCollection = currDB.getCollection(arguments.get("dest"));
    // Clear the destination collection if we are supposed to
    if (arguments.containsKey("overwrite")) {
      destCollection.remove(new BasicDBObject());
    }
    // Store the size of the collection before we start
    long collectionSize = destCollection.count();
    // Now we are ready to do the copying...
    int sourceRecordsRead = processInput();
    // Tell the user all the exciting stuff we did
    System.out.printf("Read %d source records\nInserted %d records to
        destination\n",
        sourceRecordsRead, destCollection.count() - collectionSize);
    // Now close the connection
    connection.close();
  }

  private int processInput() {
```

```
  // Figure out what our recordSize will be
  int recordSize;
  try{
    recordSize = Integer.parseInt(arguments.get("rsize"));
  }catch( NumberFormatException e){
    System.out.println("Could not parse 'rsize' as integer: using
        rsize=1 instead.");
    recordSize = 1;
  }
  ArrayList<DBObject> records = new ArrayList<DBObject>(
      recordSize);
  // Keep track of how many records read
  int recordsRead = 0;
  // Now we iterate through the source collection
  DBCursor inputCursor = sourceCollection.find();
  while( inputCursor.hasNext()){
    records.add( inputCursor.next() );
    recordsRead++;
    // If we now have recordSize records, then write it out
    if (records.size() == recordSize) {
      destCollection.insert(new BasicDBObject().append("mainArray",
          records));
      System.out.println("New record inserted. recordsRead = " +
          recordsRead);
      records.clear();
    }
  }

  // If we have something left at the end, insert it as well
  if (!records.isEmpty()) {
    destCollection.insert(new BasicDBObject().append("mainArray",
        records));
    records.clear();
  }
  // Return number of records we read from source
  return recordsRead;
}

private void processOpts(String[] args) {
  int i = 0;
  while (i < args.length) {
    if (args[i].substring(0, 2).equals("--")) {
      if ((i + 1) < args.length) {
        if (args[i+1].length() > 2 && args[i + 1].substring(0, 2).
            equals("--")) {
          // Both this argument and the next one are standard
          // arguments
          // So we just store this one and move on
          arguments.put(args[i].substring(2), null);
```

```
            i++;
          } else {
            // The next argument goes along with this argument
            arguments.put(args[i].substring(2), args[i + 1]);
            i += 2;
          }
        } else {
          // This must be the last argument, so just store it if we
          // get here
          arguments.put(args[i].substring(2), null);
          i++;
        }
      } else {
        // Arguments that don't begin with a dash and aren't
           associated
        // with another argument
        // should be ignore
        i++;
      }
    }
  }


}
```