RICE UNIVERSITY

Improving the Efficiency of Map-Reduce Task Engine

by

Mehul Chadha

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Alan L Cox

Alan L. Cox, Chair
Professor of Computer Science

[illegible]

Scott Rixner
Professor of Computer Science

Vivek Sarkar

Vivek Sarkar
Professor of Computer Science
E.D. Butcher Chair in Engineering

Houston, Texas

October, 2014

ABSTRACT

Improving the Efficiency of Map-Reduce Task Engine

by

Mehul Chadha

Map-Reduce is a popular distributed programming framework for parallelizing computation on huge datasets over a large number of compute nodes. This year completes a decade since it was invented by Google in 2004. Hadoop, a popular open source implementation of Map-Reduce was introduced by Yahoo in 2005. Over these years many researchers have worked on various problems related to Map-Reduce and similar distributed programming models. Hadoop itself has been the subject of various research projects. The prior work in this field is focussed on making Map-Reduce more efficient for iterative processing, or making it more pipelined across different jobs. This has resulted in an improvement of performance for iterative applications. However, little focus was given to the task engine which carries out the Map-Reduce computation itself. Our analysis of applications running on Hadoop shows that more than 50% of the time is spent in the framework in doing tasks such as sorting, serialization and deserialization . We solve this problem introducing an extension to the Map-Reduce programming model. This extension allows us to use more efficient data structures like hash tables. It also allows us to lower the cost of serialization and deserialization of the key value pairs. With these efforts we have been able to lower the overheads of the framework, and the performance of certain important applications such as Pagerank and Join has improved by 1.5 to 2.5 times.

Acknowledgements

This thesis would not have been possible without the constant guidance of my advisor Alan L. Cox. I would like to thank him for everything. Our countless discussions on systems design helped me get the ideas which became the central theme of my thesis. The three years I spent at grad school under Alan's guidance has helped me grow as a person. Observing his thought process while solving system problems has been an inspiration for me over the last few years. Now, when I start a new phase in my professional life, I will miss him.

I would also like to thank Scott Rixner and Vivek Sarkar for their advice and their comments which helped me to improve this thesis. Special thanks to both of them for believing in my research and their encouragement of my work.

Sorav Bansal, under whom I had my first research experience at IIT Delhi, has been my constant source of inspiration. I owe a great deal of gratitude to him for believing in me when I was just out of college without a strong research experience under my belt.

Special thanks to my Mom and Dad for their love and encouragement. They have stood besides me always and their support has given me the stength through all the ups and downs of life. Special thanks to my little brother Parth for all his love and support. I would also like to thank both of my grandmothers for all their love and blessing.

Lastly I would like to thank a lot of people who made my stay at Rice wonderful. I would like to thank my office mates Myeongjae and Conglong. Thanks in particular to Brent Stephens for his valuable feedback on my thesis presentation. I would like to thank my room mates and dear friends Kaushik, Satyakam, Rakesh and Kuldeep.

Thanks to Mayank with whom I have had a lot of technical discussions and I have always learned something through that process. Lots of thanks to Kumud for making my stay at Rice a lot of fun. Thanks to my seniors Karthik, Milind and Deepak. I value the technical discussions I have had with them which has helped me to grow professionally. Outside Rice I would like to thank my dear friends Anurag, Kartik and Atin. I wish them all the best in their future endeavours.

Contents

Abstract ii

List of Illustrations vii

1 Introduction 1

1.1 Hadoop: Map - Reduce . . . 3

1.2 Thesis Statement . . . 6

1.3 Contributions . . . 6

1.4 Organization . . . 8

2 Background 9

2.1 Data Storage and Programming model . . . 9

2.2 MapReduce Layer . . . 11

2.2.1 Map Side Execution . . . 12

2.2.2 Reduce-Side Execution . . . 14

2.2.3 Combiner . . . 16

2.3 Applications . . . 17

2.3.1 Word-Count . . . 17

2.3.2 Relational Join . . . 19

3 Related Work 22

4 Map-Reduce computation models 27

4.1 Asynchronous Design . . . 28

4.1.1 Mapper - No Combiner . . . 28

4.1.2 Reducer 29
4.1.3 Mapper - With Combiner 30
4.1.4 Implementation Details 32
4.1.5 Intuition 36
4.1.6 Limitations 36
4.2 Synchronous Design 37
4.2.1 Overview - Abstract Design 37
4.2.2 Implementation Details 39
4.2.3 Join specific optimizations 40
4.2.4 Hashmap Construction 41

5 Analysis and Evaluation 44
5.1 Experimental Setup 44
5.2 Workload Generation 45
5.3 Word Count 45
5.3.1 Application Breakdown 49
5.4 Pagerank 51
5.4.1 Size Invariant Performance 52
5.5 Join 53

6 Conclusions and Future Work 56
6.1 Conclusions 56
6.2 Future Work 57

Bibliography 59

Illustrations

1.1 Figure showing how the data from Map phase is sorted and then collected by the reducer where it is merged before the reduce function gets to process the key value pairs. 4

2.1 Figure showing flow of information from JobTracker to namenode, and finally into the available map slots 10
2.2 Output files of a mapper, where index file has pointers inside the data file. 12
2.3 The iterator fetches key value pairs from the Heap, and hands them over to the reduce operation 15
2.4 Figure showing the operations that happen when a combiner is being used in an application 16

4.1 Figure showing how the processing in combiner gets affected with the new interface 31

5.1 Word-Count performance with increasing number of reducers 46
5.2 Word count performance with 30 mappers and 9 reducers 48
5.3 Map Phase Execution time for word-count with 128MB block size . . 48
5.4 Word-Count performance break down in Map Phase for baseline Hadoop 49

5.5 Word-Count performance break down in Map Phase for our modified Hadoop 50
5.6 Pagerank performance with increasing reducers and block size 51
5.7 Reduce side Join performance with increasing size of reference table . 54
5.8 Reduce side Join performance with increasing size of reference table . 55

Chapter 1

Introduction

This dissertation focuses on improving the performance of the applications written in a distributed programming model such as Map-Reduce[1]. We redesign the task engine, the heart of a Map-Reduce platform to make it more efficient while providing a minimal interface change to the programming model. We use Hadoop, an open source implementation of Map-Reduce to demonstrate our ideas. Nonetheless, the ideas in this thesis are not limited to Hadoop, and can be applied to other platforms which use Map-Reduce or a similar higher level programming interface.

The popularity of Hadoop and Map-Reduce in general can be attributed to the simple interface provided to the programmers. This interface abstracts out the complex details of parallelization, message passing and fault tolerance. This leaves the programmer to design and implement their algorithm with map and reduce functions, which are able to implement any distributed computation[2]. The final program can be a single Map-Reduce job or a series of Map-Reduce jobs where the output of one job becomes the input to another job, and the output of the last job is the final output of the program.

The ease of programmability and reduced human effort has led to wide scale adoption of Map-Reduce and Hadoop. Google processes 20 PB of data in a day with MapReduce [1]. Facebook stores 15PB of data in their Hadoop cluster [3]. Hadoop's adoption is not limited to the high tech industries. According to the Hortonworks survey, Hadoop today is deployed by many large organizations across many industries including Healthcare, Retail, Financial Services, Government and Manufacturing [4] eg. JP Morgan [5], Visa [6], and New York Times [7].

This year completes a decade since the Map-Reduce model was first published by Google in 2004. Since then, many researchers have worked on various problems related to Map-Reduce and similar distributed programming models. Hadoop itself has been the subject of various research projects. Many researchers have studied ways of improving the job response times by improving the scheduling of new tasks and straggler tasks[8, 9]. Machine learning and graph processing algorithms are implemented in Hadoop. As these algorithms happen to be iterative in nature, there has been work to optimize iterative Map-Reduce jobs [10]. Map-Reduce online[11] redesigned Map-Reduce to become more pipelined. The pipelining works from the mappers to the reducers, and also from one job to another. This improved the efficiency of the framework, and also allowed new capabilities like online aggregation and continuous queries.

However, little focus has been given to the task engine which carries out the computations in map and reduce functions. In many applications that we investigated, we found that more than half of the time was being spent in sorting, serialization and deserialization of objects. We re-designed Hadoop to solve these problems. We propose an interface change to the Map-Reduce programming model. This interface change allows us to use data structures which are more efficient. Our Hadoop can now speed up applications up to 2 to 2.5 times. Verma *et al.* solves a subset of the problems listed above[12]. We adopt different techniques to solve these problems. The key difference is an intuitive extension to the Map-Reduce interface. This interface helps us in also avoiding the expensive serialization and deserilization, a problem which is not solved by Yang *et al.* Our interface provides opportunities for an asynchronous reduce model, which removes the need to have sorting anywhere in the entire pipeline. This interface is also designed to be backward compatible, and thus can support a reduce model which is synchronous as well. We provide an option for either the asynchronous or synchronous model through this new interface. We leave the decision of its use to the programmer. As the programming interface change is mechanical in nature, it can be automated in the future.

Native Task [13] proposes a more cache oblivious sorting algorithm in Hadoop and proposes to move the compute intensive tasks to JNI. JNI brings added complexity to the framework, as it makes the entire system unportable.

1.1 Hadoop: Map - Reduce

Hadoop is largely composed of two different components, the data store or the HDFS(Hadoop Distributed File System) and the computation engine which implements Map-Reduce which is responsible for scheduling the jobs and their respective tasks on the cluster.

Map-Reduce is a data centric programming model. Programmers have to express their computation in the form of map and reduce functions. The map function is executed locally on every node. After an all-to-all communication has taken place, the reduce is called for every key and all the values associated with that key.

Data in Hadoop is stored as a file. It also gives a block level interface, which the Map-Reduce framework uses to launch mappers on the nodes.

Abstractly the Map-Reduce framework does the following:

- Schedule mappers and reducers on available nodes.

- Allocate input blocks to the mappers.

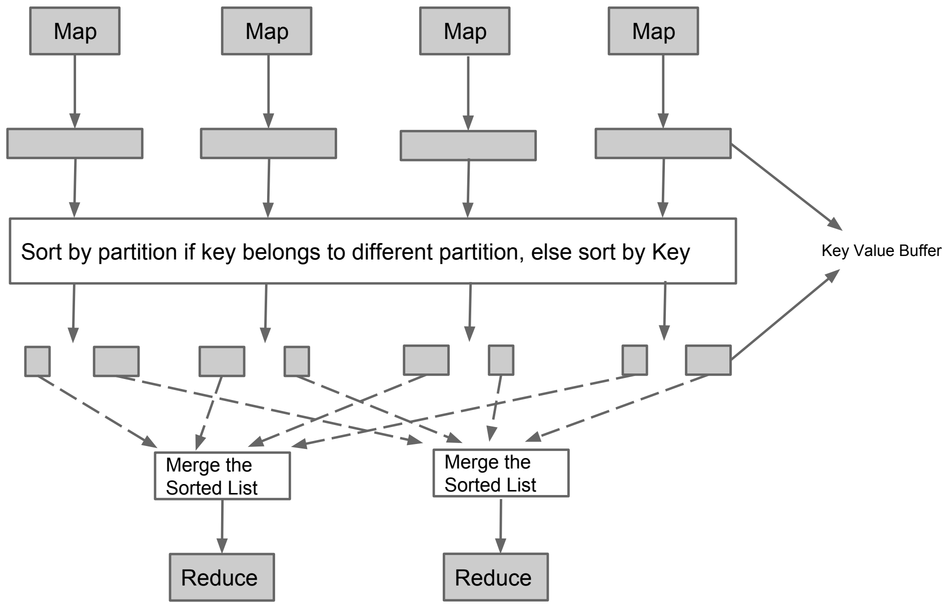

Figure 1.1 : Figure showing how the data from Map phase is sorted and then collected by the reducer where it is merged before the reduce function gets to process the key value pairs.

- Read the block and call map for every input record in the block.
- Launch the number of reducers as given by the programmer.
- On the reducer, the framework should be able to collect all the key-value pairs emitted by all the mappers that belong to its partition.
- The reducer should then call the programmer defined reduce for every key and all the associated values for that key.
- The emitted key-value pairs from the reduce phase are considered as the final output of the job. This is written in the persistent store as a file in the HDFS. Each reducer creates its own file in the output directory given by the user.

The above model defined can be used to implement many data centric programs on a large cluster. The performance of Map-Reduce applications will be highly dependent on how the above model is implemented. To describe the contributions made in this thesis we describe only the relevant implementation in Hadoop below. A detailed description is provided in section 2.2.

To maintain conciseness the following details assume the memory is infinite.

- The emitted key value pairs from the mapper are buffered in memory. A seperate data structure stores the offsets of every key value pair in the buffer and also the partition which the key belongs to. To collect all the key value pairs belonging to the same partition, the framework uses sorting where the key is the partition number. This description here is incomplete. The next point gives further motivation and also completes the description of the map phase.

- The reducer is expected to call the reduce function for every key and all the values associated with the key. Thus, every reducer should collect all the values together that have the same key. Naively, this can be implemented by collecting all the key value pairs from all the mappers and then sorting them by the key. This will collect all the values for the same key together. Since the number of mappers are more than the number of reducers in general, instead of sorting taking place both at mappers and reducers, sorting is moved to the mappers and the reducer just does the merging of the sorted lists. To complete the description we left above in the map phase, the map sorts the key value pairs by the partition number when they belong to different partitions and sorts them by the key when they belong to the same partition.

- The reducer merges the sorted key value pairs that it has received from the mappers. The framework then iterates over the merged list, calling reduce for every key, with the list of values associated with the key.

One of the side effects of the implementation above is that the output from all reducers is sorted by the key. Since the keys are partitioned, the output is not fully ordered. It is difficult to argue whether a sorted output was a desired effect. Nonetheless, in this work we argue that by relaxing the constraint of having a sorted output, we can improve the performance of applications in general. Since most of the times the output of the Map-Reduce applications is consumed by other applications, we believe we can do away the need to have a sorted output.

1.2 Thesis Statement

The efficiency of any Map-Reduce implementation is directly related to the mechanisms involved in processing the data. These mechanisms are directly related to the programming interface that is provided to the programmers. This thesis shows that a change in the programming interface allows us to improve the internal mechanisms. The new interface is simple and intuitive. With the use of efficient data structures, the overheads are lowered. In the applications that we evaluated, our results show an improvement up to 2.5 times.

1.3 Contributions

At a high level this thesis presents a more efficient task-engine for Map-Reduce. The higher efficiency is achieved by removing or reducing the current overheads like sorting, serialization and deserialization. An efficient task engine results in an improved performance for Map-Reduce applications. Our evaluation show a speedup from 1.5

to 2.5 times.

- We introduce two different models for computation in a Map-Reduce based system. Both our models do not require sorting of the key value pairs. The first scheme adds the capability to make the reducer asynchronous with respect to the map phase. Our second model is designed for certain special class of applications such as joins. In this model of Map-Reduce, we retain the existing property where the reducer is synchronous with respect to the map phase. In chapter 4 we give a detailed description of both the models.

- We change the programming interface for Map-Reduce. This interface change adds the capability to make the reducer asynchronous, and removes the need to have sorting anywhere in the pipeline. This interface is backward compatible with the older interface, and thus our second scheme of synchronous reducer works with this new interface as well.

- In the asynchronous reduce model, the computation of key value pairs can start as soon as the first output is available from the mapper unlike the synchronous model where the reducer can start the actual computation only when all the key-value pairs are collected together. The framework also makes the combiner an in memory operation. This requires no change at the programming interface. The application performance improves, as the expensive overheads of sorting, serialization and deserialization are removed.

- The second model for the Map-Reduce engine is specially designed for applications like Join. This model is based on a hash join based approach used in parallel databases. We use a novel approach in collecting the key value pairs through the use of hashing. Instead of partitioning data on the key and sending data across, we create hashmaps in the map phase, shuffle the hashmaps and then join keys across these hashmaps in the reducer. Building hashtables on the mappers is less expensive than creating them on the reducers. This is because the mappers are usually more in number than the reducers in the cluster. The key idea here is based on the observation that the insertions are more expensive than the lookups.

- Java hashmaps are known to exhibit poor performance and are not designed to be shared across nodes in a cluster or within different applications in the same node. We implement a high performance hashmap based on cuckoo hashing [14]. The construction of this hashmap is based on MemC3 [15]. We construct this hashmap on an mmap interface in java. This allows the hashmap to remain in memory, and yet be shared across different services.

1.4 Organization

The remainder of this thesis is organized as follows. In Chapter 2 we describe in greater details the current Hadoop Map-Reduce system. We explain the internals with the help of a few Map-Reduce applications. Chapter 3 covers the related work. Chapter 4 covers the design of our faster Map-Reduce models. In chapter 5 we present the results and an evaluation study for our faster Map-Reduce design. In Chapter 6 we present the conclusions along with the future work.

Chapter 2

Background

This chapter presents the design and implementation details of the Hadoop Map-Reduce based system with the focus being on the task engine. This chapter first presents an overview of Hadoop's data storage system and the Map-Reduce programming model. It then delves into the task internals in both the map and reduce phases. This chapter then explains Word-Count, Pagerank and Join which are popular Map-Reduce applications. These applications are used as examples throughout the remainder of this thesis.

In this chapter we understand the overheads associated with the Map-Reduce task engine. We also look at these applications from the programming point of view, to understand how the programming interface can be extended.

2.1 Data Storage and Programming model

Any Map-Reduce based system like Hadoop can be divided into two components - the storage layer and the computation layer. In hadoop the storage layer is the Hadoop Distributed File System. HDFS provides a file based interface like a traditional Unix file system to store data. As HDFS is a distributed file system, data is distributed across the nodes in the cluster. The granularity at which each node stores data is a block. These blocks are replicated across the cluster for fault tolerance and can also be used for I/O parallelism when two or more jobs are using the same input file.

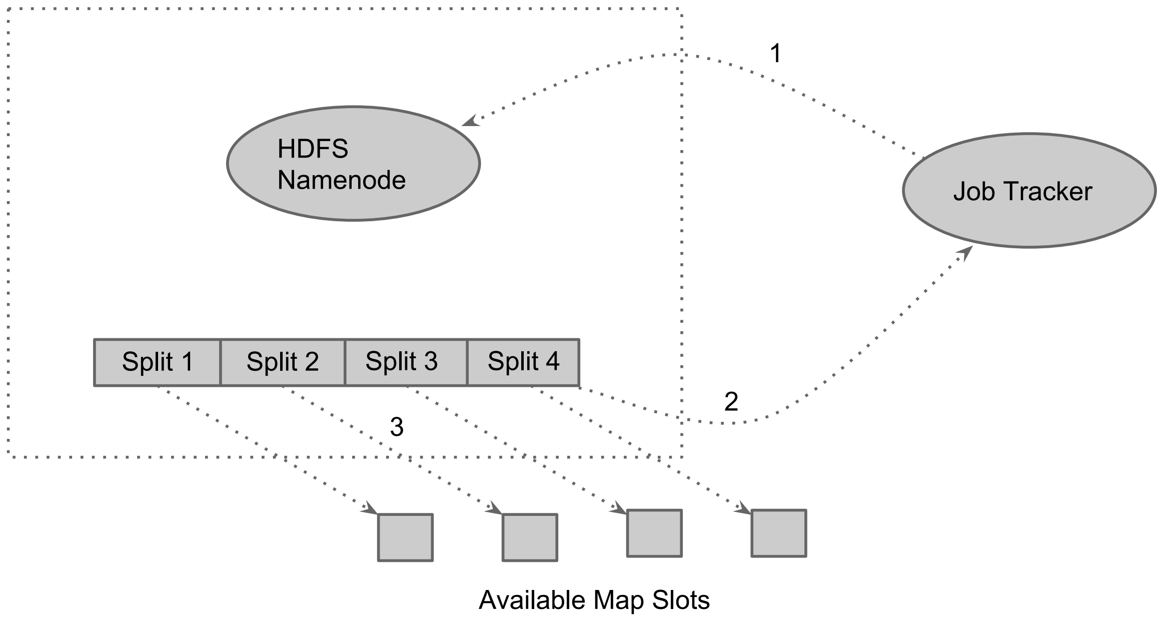

Figure 2.1 : Figure showing flow of information from JobTracker to namenode, and finally into the available map slots

Hadoop uses a concept of input split which can take a value less than or equal to the size of the block. The number of mappers launched is equal to (Input Size/Split Size). Having a small input split size affects the performance as it makes poor utilization of the available disk bandwidth. A large input split size however can reduce the available CPU parallelism in the cluster. Input split size ranges from 4MB to 128MB depending upon the type of application. Input split size if chosen by the Map-Reduce programmer. Figure 2.1 shows the interaction of the system with the hdfs.

The programming model is inspired by the lisp functions map and reduce. At an abstract level the programming interface requires the programmer to implement two functions map and reduce. The map function runs locally on every node. It emits intermediate data in the form of a key and a value. The reduce function then aggregates all the values for every key. The aggregation here refers to the computation that is performed on all the values. At the implementation level, Hadoop and other similar Map-Reduce systems which are implemented in an object-oriented language, there exist a mapper and a reducer classes which are extensible. The programmer overrides the map function in the mapper class and the reduce function in reducer class to implement the map-reduce logic. Both the classes have setup and cleanup functions which can be overridden. The Function "setup" is typically used to create data structures for later use by the map and reduce functions. The Function "cleanup" is used to do any post processing after all keys have been processed.

2.2 MapReduce Layer

Map-Reduce is the layer responsible for the computations. This layer implements the programming model discussed above. It is also responsible for scheduling the mappers and the reducers. setting up the inputs for both the map and reduce functions is done by this layer. To begin with, this layer obtains the block locations for the input files and then schedules the mappers. The scheduler takes into account the proximity of the data block to the node where it can be scheduled. Highest priority is given when the data block can be scheduled locally. This plays an important role in the performance of the overall application as the aggregated disk bandwidth at the nodes is much larger than the bisection bandwidth of the network.

The programmer specifies the path of the input files which are to be used as an input to the Map-Reduce program. It is also the responsiblity of the programmer to specify the program which will read the raw data in the file, and convert it into a key value pair. It is this key vlaue pair which is given as an input to the map function. In Hadoop this is called as a record reader. So, for example in the text input file every new line is considered as a record. The input key to the map in such a case is the

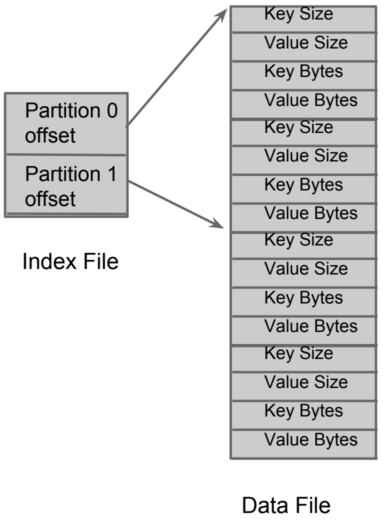

Figure 2.2 : Output files of a mapper, where index file has pointers inside the data file.

offset in the file, and the value is the entire line encoded as a String object.

The key value pairs that are emitted by the mapper are then partitioned and sent to their respective reducers. Every emitted key and all its associated values are given as an input to the reducer. It is the responsiblity of the framework to collect all the values of the key and call the reduce function. We explain the internal working both in the map task and the reduce task in the next section.

2.2.1 Map Side Execution

As explained previously the map function is called for every input record in the input split or the block allocated to the mapper. The emitted key value pairs from the map function are serialized into a buffer. The memory for this buffer is allocated during the initialization of the mapper. The size of this buffer is controlled by the configuration parameter io.map.buffer. While serializing the emitted key values pairs, the key is also hashed to find the partition that the key belongs to. This information along with the key and value offsets are maintained in a seperate data structure. As hadoop uses the technique of sort-merge-join to collect all values of the same key, the final output of the map task needs to be sorted. As the buffer size is limited, everytime it is filled up, this buffer is sorted and spilled to disk. Hadoop uses quick sort to sort the buffer. The comparator for the sorting algorithm first checks the partition number of the two keys being compared. If they belong to different partitions then the partiton numbers are compared. If the keys belong to the same partition, then the keys are compared together. Thus at the end of the sort phase each partition will be collected together, and keys inside each partition will be sorted by key. This brings all the keys which are the same together, which is what is required for the reduce operation. This operation of sort and spill creates two files, a data file and an index file as shown in figure 2.2. The data file contains the serialized key-value pairs, and the index file contains pointers or offsets for every partition in the data file. At the end of the map phase there are two possible scenarios. If the buffer size was large, no sorting and spilling would have taken place. In this case the aforementioned sorting is carried out. If the buffer size is small, we will end up with multiple sorted buffers which now need to be merged into one single buffer with all the partitions and all partitions individually sorted. This is done using a standard priority queue implemented as a heap. For every partition a priority queue is constructed. The elements of the heap are the sorted buffers for that partition. Key values pairs are extracted from this heap data structure, which will be in sorted order as that is the property of this data structure. Assuming there are m sorted buffers in this tree and n keys in all, the cost of merging these sorted segments will be O(n log m). To reduce m, hadoop uses a parameter called the maximum merge factor. If there are more than m segments to be merged, then m of them are first merged into a sorted buffer and then that sorted segment is added as an element in the priority queue. This operation reduces the elements in the heap by (m-1). At the end of the merging operation, a data file and an index file are the expected output as explained previously.

After all the records have been exhausted by the mapper, the next stage in the pipeline is a combiner which is also a local reduce operation. This is an optional phase in the entire execution. Section 2.2.3 explains the internal working of the combiner and its use case after explaining the reducer since the idea of a combiner is the same as a reducer.

After the map phase is completed, it is communicated to the resource manager in the new version of Hadoop, or the job tracker in the older version of Hadoop. It is then the responsibility of the node manager or the job tracker to notify the reducer to fetch the map output. The JVM responsible for the mapper now retires, and the fetch operations from the reducer are fulfilled by the task tracker or the nodemanager(YARN), which is local to every node. When a particular reducer sends the request for fetching map output, the index file is used to index into the data file, and then the data for that partition is sent to the particular reducer.

2.2.2 Reduce-Side Execution

The reducers are scheduled with respect to completion of mappers. This is dependent upon the configuration parameter called slow start. When the slow start value is 0, the reducers will be scheduled at the earliest. The scheduler does not schedule all the reducers in the begining because the reducers are waiting for the mappers to finish. If all the slots are allotted to the reducers, it will lead to a deadlock. When the value is 1, all the reducers will be scheduled only after the map phase is completed. The

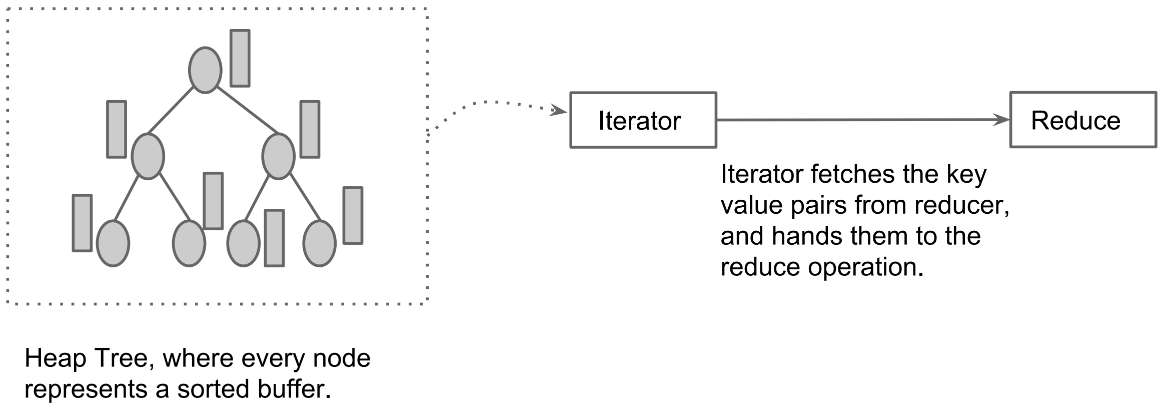
Iterator
Reduce
Iterator fetches the key value pairs from reducer, and hands them to the reduce operation.
Heap Tree, where every node represents a sorted buffer.

Figure 2.3 : The iterator fetches key value pairs from the Heap, and hands them over to the reduce operation

reducer starts by fetching the map outputs as and when the mappers have completed. As the map outputs are already sorted by keys, a merge needs to take place. This is similar to the one that we explained in the mapper phase. It uses the same logic of using a priority queue as explained previously. The same merge factor is used here, and an iterator is created for the heap. The only key difference between merging in the map phase and in the reduce phase is that we don't need to create a finally merged sorted buffer. This is because the reduce function fetches the key value pairs from an iterator, and we can always create an iterator over the heap data structure. The step-by-step procedure is as follows:

- Create an iterator over all the sorted buffers if the number of sorted buffers is less than the merge factor. Otherwise, as the number of received map outputs reaches the merge factor, merge them into a single sorted buffer and add this element in the heap tree.
- Retrieve the first key from the iterator, deserialize it and call the reduce func-

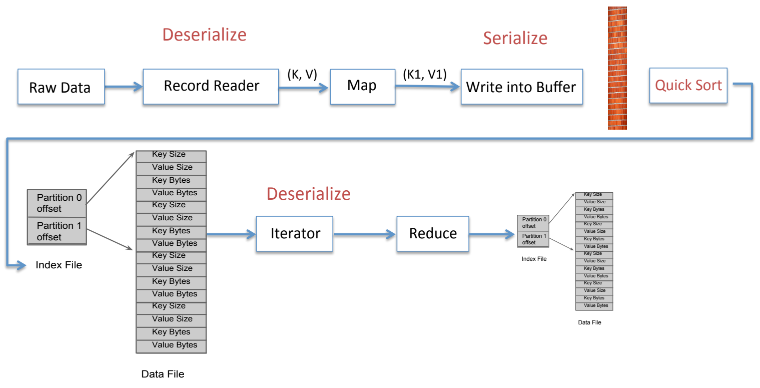

Figure 2.4 : Figure showing the operations that happen when a combiner is being used in an application

tion, passing the deserialized key and an iterator on the values as the second argument.

- Inside the reduce function, for every new value that is asked by the iterator, the framework compares the key with the previous key. If they are equal, then the value is deserialized and returned by the iterator. Otherwise the reduce function returns as the iterator returns null.

- The framework picks the next key and the whole procedure repeats till all the key value pairs are exhausted.

The above procedure is also explained more clearly in the figure 2.3.

2.2.3 Combiner

We now explain the use and the implementation details of the combiner in detail. There are many advantages of doing a combine or a local reduce operation before sending data to different reducers. First, it greatly reduces the communication overhead. Since all the key value buffers are sorted in one pass, duplicate keys exist as many times their value exists. If the application is such that all the values for the key can be reduced to a single value, the combiner not only reduces the space occupied by the values but also by the keys. However the combiner can not be used in all the applications. For example in the next section we explain the Word-Count which uses a combiner. On the other hand the Join application does not benefit from having a combiner. Having a local reduce also reduces the computation time at the reducer, since this computation is being done in parallel by all the mappers. The combiner uses the index and the data file created after the map phase to do the local reduce. For every partition, an iterator is created which iterates over the sorted buffer returning keys and values to the iterator in the reduce function. This works similar to the design as explained in the reducer. The only difference is the underlying iterator which in this case is implemented over the sorted buffer, while in the case of the reduce phase it was over the heap data structure. For combiner the framework applies deserializer to all the keys and values, and the output of combiner undergoes sorting again. To summarize the combiner improves the performance by reducing I/O and introducing more parallelism. This is shown clearly in figure 2.4.

2.3 Applications

We now explain how Word-Count and Joins are implemented in hadoop.

2.3.1 Word-Count

Word-Count is one of the simplest programs implemented in a mapreduce programming model. Input to the Word-Count program are a group of text files stored in the HDFS. The record reader for a text file reads a line as a record, and calls the map function on it. As shown in code listing 2.1 the map function uses StringTokenizer to break this line into words. Every word is then emitted as the key, with the key of type Text and value of type Intwritable. Here Text and IntWritable are wrappers over String and Integer respectively. These classes implement functions for serializing and deserializing their respective values. The value of integer is set to 1 for all keys. By virtue of the design of the Map-Reduce framework, all keys will be sorted and partitoned. Every reducer will fetch the mapper output belonging to its partition, merge the outputs and call the reduce function for every word and all its values. Inside the reduce function, all values are summed up and the final sum is the number of occurrences of that word in the input data. The combiner also vastly improves the performance of the Word-Count application by reducing the I/O overheads. The code listings 2.1 and 2.2 shows the code for the mapper and reducer class respectively.

Listing 2.1: Mapper Class implementation in Hadoop for the application Word-Count

```
class Map extends Mapper {
  Text word = new Text();

  public void map (LongWritable k, Text v, Context C) {
    String s = v.toString(); // Deserialize
    StringTokenizer t = new StringTokenizer(s);

    while(t.hasMoreTokens()) {
      word.set(t.token());  // Serialize
      C.write(word, 1);
    }
  }
}
```

Listing 2.2: Reducer Class implementation in Hadoop for the application Word-Count

```
Class Reduce extends Reducer {
  public void reduce (Text k,  Int[] v, Context C) {
    int sum;
    While (val:v) {
       sum += val;
    }
    C.write(k, sum);
}
```

2.3.2 Relational Join

Joins is a popular application where data from differenct sources are combined. It is typically used in many data analytic operations. The most common type of joins supported by hadoop mapreduce is an equi-join. In a typical one-to-many equi-join operation, a key(primary key) which uniquely identifies a row from one source, is combined with the same key in a different source where this key(foreign key) identifies multiple rows. There are primarily two ways to do joins in a map-reduce environment, namely map-side join and a reduce-side join. Map side join is a restrictive technique where one of the tables must be able to fit in memory for the join to take place. The larger table is already partitioned as it is stored in the HDFS.

A more general technique is a reduce side join where the key to be joined is emitted in the map phase with the value being the attributes which need to be joined from both the tables. Since all the keys are mapped to the same partition, a join can be implemented in the reduce function. The framework will ensure that the reduce function receives all the values associated with the key. However, since we need to join the key from one table to another we need to know explicitly which values belong to which table. This information can be encoded in the value itself, in which case before the join operation takes place, all the values have to be retrieved from the value iterator, stored in memory as a deserialized java object, and then in another pass the join will be done which will be written to the hdfs. This operation is expensive since it starts to consume more memory in the form of java objects, and increases the cost of garbage collection [16]. If however, the values can be retrieved in an order, where the first value received from the iterator belonged to the primary table, and all the following values belonged to the other table, then the costs can be reduced. This is because only the value for the primary key will remain in memory, while all the following values fetched would be joined with this value from primary table, and the output will be written to the HDFS. Once the join has taken place, and the output emitted, this value can be garbage collected. This technique improves the performance of joins as shown by blanas et al. [16] The second technique uses less memory, since deserialized java objects occupy much larger memory than the serialized objects stored in memory. This is implemented in map-reduce using a simple idea. Instead of the the key being emitted out by the mappers, a composite key is used. The actual key is appended by a tag. When the table for this key is a primary table, the tag value is 0. On the other hand it is 1, when it belongs to the other table. This simple idea ensures that when the keys are merged in the reducer, the key with tag 0 comes before the keys with tag 1. The first value retrieved in the reduce will belong to the primary table, and all the values following will be from the other table. However, a small change is required while partitioning the keys. Since we need to find the partition only on the actual key and not on the tag, we extend the partitioner class and extract the key from the composite key, find the partition number and return it as a result. This will ensure that all the composite keys with the same join key reaches the same reducer.

Chapter 3

Related Work

The main contribution of this work are the optimizations applied to the Hadoop Map-Reduce based systems. This work presents at a higher level, two different models for Map-Reduce computation, namely the asynchronous reduce, and synchronous reduce model. As part of these models, we propose a novel extended interface, which is intuitive and requires minimal programming effort over the existing model. In the asynchronous reduce model this interface removes the need to have sorting anywhere in the entire pipeline. This interface facilitates an asynchronous reducer and also an in-memory combiner. Our synchronous reduce model is designed for a particular set of applications like joins. Our contributions for this work are faster hashmaps based on cuckoo hashing and the novel ways in which the joins happens.

Map-Reduce is not the only distributed programming model. Dryad [17] and DryadLINQ [18], like map-reduce is a data parallel programming model and runtime that supports a model of acyclic dataflow graphs. These programming paradigms help write distributed programs as a graph computation. Our work solely focuses on improving the performance of the Map-Reduce engine. The ideas used are not specific to Map-Reduce environment, and can be used in other distributed programming models as well. The earliest work in mapreduce based systems which advocated changes to the application interface was map-reduce-merge[19]. The work proposed an additional merge step which merges the data from heterogenous datasets. The writes done in the reduce step are sent for merge. Our work provides interface changes which help optimize the map-reduce programs, and do not provide additional functionality like the work done in map-reduce-merge.

The following related work falls in the area of applications running on a cluster, and which are iterative in nature. It was soon realized in the research community that for large scale iterative jobs, none of the existing high level distributed programming models such as map-reduce or Dryad could perform well. One of the reasons for this behaviour was the materialization of the output to the disk. Most of the data intesive machine learning based applications such as Pagerank [20], HITS[21], recursive relational queries[22], clustering, neural network analysis happen to be iterative in nature. Haloop, [10] was the first step of modifying hadoop to use it for iterative processing. It proposes a new programming interface to express iterative processing. It modifies haloop's task scheduling engine to be aware of iterative jobs such that data can be reused across iterations. It also implements caching of loop invariant data across iterations, which further optimizes the performance. Spark [2] was designed from grounds up to solve the problems of poor performance for distributed applications which are iterative in nature. Spark provides much broader interfaces to the programmers and thus can be used for a variety of applications, and not just map-reduce style applications. Spark proposed a novel abstraction of RDD's(Resillient Distributed Datasets). These are fault-tolerant, parallel data structures which can be shared. There are a rich set of operators that can be applied on RDD's. Fault tolerance is provided by tracking the lineage of a RDD. This is tracked by logging the operations that were done to construct the RDD instead of materializing the RDD's to disk. Thus in scenarios where failures do not occur more often, spark can provide much higher performance. As these RDD's are designed to be shared across iterations, data often does not need to be serialized and then deserialized to reconstruct objects.

The following related work is in the area of scheduling. Scheduling of tasks and jobs affects data locality and thus has a considerable effect on performance. Matei et al. [23] address the performance problems in heterogenous environments like Amazon EC2. The work shows how the scheduler in hadoop can cause severe performance degradation in virtualized environments. They propose LATE(Longest Approximate Time to End), which is highly robust to heterogeneity. The key idea is to use a heurestic to speculate the task which will take the longest to finish, and then execute that task speculatively. Matei et al [24] then address the problem of sharing of the hadoop cluster between multiple users. They designed a fair scheduler for hadoop. The techniques developed has two key contributions which are delay scheduling and copy-compute splitting. Delay scheduling tries to optimize for data locality by delaying the task, while also not letting the job to starve for too long. It uses a relax queuing policy to make jobs wait for a limited time if they can not launch local tasks. The paper also proposes to split the I/O intensive(Fetching Map Outputs) operations from the CPU intensive reduce operations By splitting the two operations, throughput of the system improves as I/O and CPU can now be overlapped.

Condie *et al.* [11] designed and proposed Hadoop Online Prototype(HOP), an alternative map-reduce architecture in which intermediate data is pipelined between operators. This new architecture allows online aggregation[25], where initial estimates of the results can be provided several order of magnitudes faster than the final result. The new architecture also allows to use continuous queries, where new data is analyzed as it arrives. Pipelining, instead of materializing data on the disk also improves the performance of the application as it increases opportunities for parallelism, improve utilization, and reduce response time. Our work unlike HOP does not provide any new functionality like online aggregation or continuous queries. The focus of our work is to reduce the task completion time. Our work on asynchronous reducer, is orthogonal to the online aggregation work, as we are able to maintain the state of the previous computation providing an interface change which is intuitive to the programmer. Online aggregation, although applies the reduce operation early, but it only applies to the data when it reaches a certain size. This size is provided by the programmer which is called as a snapshot. These snapshots are then written to the hdfs. The intermediate states between these snapshots is not preserved, and every snapshot creates its own state. Our work can be applied in this work, such that every snapshot builds from the state left from the previous snapshot. Thus the last result of the hdfs snapshot, will contain the result as if online aggregation did not exist. As explained previously, we do it using an intuitive interface change to the map-reduce programs.

Verma *et al.* [12], like our work also propose the asynchronous reducer model. This work also solves the same problem which is addressed in our work, which is to start the reduce operation on the data as it becomes available. The key difference between the two work is the approach taken to solve the problem. To be able to run reduce on the same key multiple times with a different set of values requires stateful computations, ie the state from the previous computation should be stored and be retrieved when processing the same key again in the future. In this work this problem is solved by the programmer explicitly saving the state in a tree map data structure, and then retrieving it later. This requires the programmer to create explicit objects of the state, and then storing them in the tree map. Our approach makes the state implicit at the programming interface itself. We use the fact that java is an object oriented language, where code and data can exist together. In our approach the programmer has to move the variables from the reduce function to class members. The programmer also splits the reduce function. One function is responsible for the computation, while the other is responsible for the final write operation into the hdfs. We claim that because of the mechanical transformation from the original interface to the new interface, in future this step can be automated by the compiler. This new interface also facilitates an in-memory combiner. The internal framework creates new objects of this class for every new key it observes, and for the keys which have already been observed, reduce operation is run on the object. In applications like joins where asynchronous reduce model is not suitable because of the need to have large intermediate state, we propose to solve this problem differently. We propose to use the existing concept of synchronous reduce, but we still get improvements in performance, by avoiding sorting in various phases of the map-reduce computation by developing a high performance hashmap which can be shared trivially between nodes, and between the processes of the same node.

Native Task [13] studies the task engine, and proposes sorting algorithm based on a cache oblivious model. It also proposes JNI for much of the compute intensive tasks in hadoop. Although JNI does help in improving the performance, but it brings along other drawbacks associated with JNI in general, and its use in hadoop makes the entire solution unportable.

We also study joins in this work, and optimize the reduce side join. Joins has been well studied in the map-reduce framework. [16, 26]. Sopranos *et al.* optimizes the joins at the application level, exploiting the Map-Reduce framework. By changing the internal Map-Reduce task engine, we can engineer joins for higher performance. This does not require our new interface desgined for asynchronous reducer, but since our interface is backward compatible, it can be used for joins too.

Chapter 4

Map-Reduce computation models

Many Map-Reduce programs spend significant time in sorting, serialization and deserialization of data. We solve this problem by creating a novel interface which is an extension to the original Map-Reduce programming model. This chapter shows how this new interface supports map reduce style computations without the need to have sorting, serialization and deserialization of data.

If we look closely at the different type of reduce functions in Map-Reduce applications, they can be divided into two categories. Typically in a reduce function, we have an accumulator whose state is updated with every value passed into the reduce function. Our division of reduce functions into two categories is based on the size of this state variable. We refer to this two categories as large and small. As Pagerank aggregates or folds multiple value types into a single value type, the size of the state is small when compared to a Join application where multiple values of a key are not folded into a single value. Here every value of a key is part of the state. For applications where the size of this state is small, we use an asynchronous model, on the other hand when the state consumes a large amount of memory we use synchronous.

We first delve into the details of the asynchronous model for Map-Reduce. We then explain the problem with applications with large state with this design taking the application Join as an example. Finally we explain the design of our synchronous model. For both the models, an explanation is given at an abstracted level first to develop an intuition, and then we give the implementation details where specific optimizations are also explained that help us in achieving a better performance.

4.1 Asynchronous Design

Our asynchronous implementation spans across both the mappers and the reducers. As combiners are based on the reducers themselves, we first explain our changes to mappers when the combiner is not present. We then go on to explain our reducer and finally the mapper with a combiner.

4.1.1 Mapper - No Combiner

Let us first look at a Map-Reduce application without a combiner. Since there is no computation required on any key and value after it has been emitted from the map function and before it enters the reduce function, we can place them directly where they will be fetched by the reducer. As reducers fetch their respective partition from every mapper, we create these partitions in the form of buffers in the initialization phase of every mapper. On every write done in the map function, the partition number of the key is found, and the key and the value are serialized and appended into their appropriate buffer. This process keeps happening until all the data is exhausted for the mapper. In the end we have buffers which belong to every partition, which can now be fetched by the reducers.

The key difference here with respect with the existing hadoop design to the is that we serialize the keys and values directly into their respective partition, whereas in the original design all the keys and values are serialized into a single buffer. Since the original design requires all the same keys to be together, there is no advantage to creating a different buffer for each partition. Since each individual buffer will have to be sorted, hadoop uses a single buffer which is sorted using the partition number as the sort key when the keys belong to different partitions and uses the key itself as the sorting key when comparing the keys from the same partition. In the end there exists a single buffer and the indices act as pointers to the individual partitions.

Since, no sorting takes place in our design, the performance of the applications which are not compute intensive in the map phase improves siginificantly.

4.1.2 Reducer

The input to the reducer is the key and the list of all values associated with this key. Hence the reducer has to fetch all the map outputs, and then merge them. Merging here refers to the collection of all the values for every key together. This is required since the reduce function processes all the values for every key at once. We explain in detail how this merging happens in section 2.2.1. A reduce function is inspired by the higher-order function "fold" in functional programming. A typical reduce function looks as shown in code listing 2.2. The beginning of the function is where the initialization of the variables happens. Then comes the loop wherein the iterator keeps fetching the next value. Inside the loop computations are performed. These computations update the state of the accumulator variable. For example, in word count this happens to be an integer which is initialized to 0 and then every value is added to it. The loop ends when the iterator has exhausted all the values from the particular key. The next step is where some other computation on the accumulator can be performed. This step is optional, and might not be there in all applications. The next step is where the final output is emitted from the reduce function in the form of a key and a value. This is a write into the Hadoop Distributed File System, and is considered as an output of the job. Since the reduce function operates on all the values of the key at once, the reducer has to wait until it receives the output from all the mappers. Thus the slowest mapper determines when the reduce function can be invocated on the keys and their values. After all the keys and values have been collected and merged, the framework performs the reduce operation one key at a time. We break this rigidity in the model by extending the Map-Reduce programming interface. The novel interface allows the keys and values coming in any order to be given as an input to the reduce function. This computation happens on only the particular value. This computation is the same which happens in the loop above. The computation changes the state of the accumulator associated with the key, to which this value belongs. We explain in the implementation how this state is stored and retrieved for every input. When all the values are exhausted for all the keys, the optional computation is performed for every key, along with the final write into the Hadoop Distributed File System.

4.1.3 Mapper - With Combiner

As explained in 2.2.3, a combiner is a local reduce operation that is executed on every mapper. Combiners use the same programming interface as reducers. Every key and all its associated values are passed as arguments to this function. We first review the operations performed in the current system, and then explain the problems that we solve.

- The mapper emitted keys and values are serialized into a buffer.
- Sorting is used to collect the keys together.
- At the end of the mapper, an index file and a data file are created.
- If the combiner is present, then for every key and its associated values, the function combine is called.

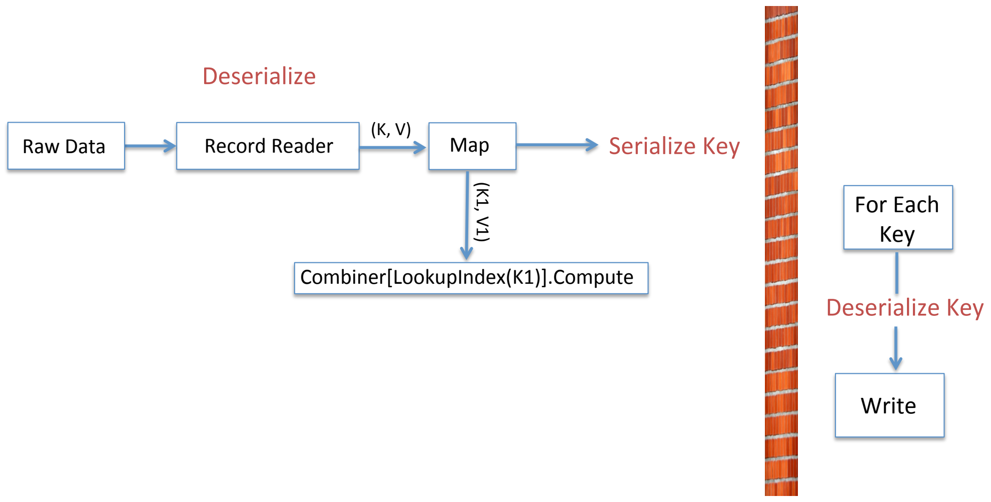

Figure 4.1 : Figure showing how the processing in combiner gets affected with the new interface

- After the computation is performed on all the values in the loop, a write will be done. The write takes place locally in an in-memory file. These local writes will then be fetched by the reducers.

There are two problems here. First, sorting is an expensive operation which is used to collect all instances of the same keys together. The second problem is the serialization and deserialization that has to take place. Serialization takes place when the writes from the map function are done into the buffer, and deserialization happens for every key and value passed to the combiner. The second problem is a hard problem to solve for the following reasons.

- Instead of serializing the keys and values, it is possible to keep them as objects in the heap. However, since these objects tend to be large in number, the cost of garbage collection goes up, and hence leads to poor performance.

- The second problem is not very evident, but can be clearly explained from the snippet of map code for WordCount in code listing 2.1. In the map function, the writes are done using the same object as the value parameter in context.write. Since the object is being reused, the older values will be lost if they are not serialized. The object can not be cloned by the bottom layers responsible for maintaining these objects, as Java supports only shallow copy for cloning. For a deep copy, the programmer has to implement the appropriate interfaces. The only solution therefore is for the programmer to create a new object everytime if it needs to be emitted. Mutating objects will therefore not be allowed. As this constraint can lead to bugs in the applications, this problem needs to be solved in a different way.

We utilize the same interface that we developed for an asynchronous reducer. On every write from the map function, the key and value objects are passed directly into the combine function. A state is maintained for every key, which is updated when a new value for that key is emitted by the map function, and hence is input into the combine function to change the state. This step is shown in figure 4.1.

4.1.4 Implementation Details

In the mapper without the combiner, all the writes happen into a file. This file is backed by memory itself, instead of the disk. We use a tmpfs file system as a mount point for this file system. Thus, for every partition there exists a file in the tmpfs, and the key-value pairs are serialized into this file. When the mapper communicates to the central node of the completion of the map phase, these in-memory files are copied to the disk. The files in memory need to be written to the disk, as Map-Reduce follows the design where the reducers fetch the results from the mappers. Since multiple jobs are being scheduled, every node can be asked to schedule mappers without the reducers being scheduled. This constraint requires that the mapper's outputs are written to disk. By writing into memory for every write performed in the map function, we ensure high write throughput is maintained. In the end, when writes are done to the disk, this is handled by the operating system kernel as a file is being copied from one file system to another. Java provides a wrapper over the system call sendfile, where a file can be copied without reading its pages into user space. This optimization is analogous to batching, as the writes to the disk take place only once during the entire operation.

4.1.4.1 New Reducer Interface

Our new reducer interface exploits the nuances in the object-oriented style of programming. Although the reduce function is inspired from the higher-order function "fold" in functional programming, it is implemented in Hadoop in an object-oriented manner. In the current Map-Reduce systems, there exists a reducer class, in which a reduce function is overridden by the Map-Reduce programmer. Since all the values of the keys are provided at once, and the output in the form of writes is done in the reduce function itself, there is no side-effect on the state from one key to another. On every reduce function, state is created and outputs are performed. If we want to provide values to the keys, as and when they come we need a way to create state, store it, retrieve the appropriate state belonging to the input key, and then perform operations on this state. To do this efficiently and elegantly, we exploit the fact that code and data are integrated in an object-oriented programming platform. We create an object of class type reducer, for every new key seen. This makes side-effects across keys impossible. If the algorithm requires some global state independent of the keys to be updated, the interface allows the global variable to be used as a static member of the class which can be updated on every value seen independent of the key it belongs to. Since the final output can only be emitted after all the values have been processed, we break the reduce function into 3 different functions. The first function "init" is called when the key is seen for the first time, and this is where the initialization takes place. The second function "compute" is the compute only function where the state is updated. The last function "write" is where any computation required on the final state is performed, followed by the write operation. We use a high performance hashmap to find the object of type reducer for every input key. For the first appearance of the key, an object is created and the init function is called on it. This object is stored in an array, and its index is used as a value in the hashmap. For every next value associated with this key, the object is retrieved from the array, and only the compute function on this object is called. When all the keys and values are exhausted, for every key the "write" function is called. Code Listing 4.1 demonstrates how the WordCount is used in our new interface. We explain the construction of this high performance hashmap later in this chapter in section 4.2.4.

Listing 4.1: Reducer Class implementation for WordCount with the new interface

```java
public class Reduce extends Reducer {
   int sum = 0;
   public void compute (Text k,  IntWritable[] v,
      Context C) {
     for (IntWritable val : v) {
         sum += val.get();
     }
   }

```

```
    public void write(Text k, Context C) {
        C.write(k, sum);
    }
}
```

Thus, the programmer has to modify the reducer class for their Map-Reduce application. Since the original reduce function has the state variables declared in the reduce function itself, we need to move them outside and make them as instance variables such that their state is preserved for every computation that is performed on them. Moving all the variables outside would cause high memory consumption, since the objects will be created for every key. Thus, only the variables which are required to maintain state should be used as instance variables. Since most of the operations performed are mechanical in nature, we claim this program translation can be automated using a compiler.

To summarize, in the reducer, for every mapper output that is fetched, we iterate over all the keys and the values, creating a new reducer object if the key is seen for the first time, or else retrieving the object from an array and calling the compute function with the new value as the argument. Finally the write function is called on every object in the array. Once the writes are done, the references to the objects are lost, and hence memory is freed through garbage collection.

4.1.4.2 Combiner

The combiner also uses the same idea as used above in the reducer. We create the combiner object everytime we see a new key. Our implementation of combiner, where the value is consumed by the combine function as soon as it is created, removes the need to serialize and deserialize the object. However, there needs to be an object associated with every key that will have its state updated. Since in real applications there are more values associated with every key, in a block of 64MB-128MB the number of keys will be a fraction of the number of values. Thus, this solves the problem of creation of too many objects as stated above in As every object created is immediately consumed by the combiner, and used to update the local state of the combiner object, it also solves the other problem where objects being reused in map no longer is a problem. Thus, no expensive serialization and deserialization of objects takes place, and with a minimal interface change, we are able to efficiently do the combiner.

4.1.5 Intuition

There exists a fundamental reason why the asynchronous model is scalable and achieves good performance. When data grows, the rate of growth of number of values is greater than the growth of the number of keys. As the state associated is proportional to the number of keys, processing data on large number of nodes becomes possible. For instance, the amount of data Facebook collects over a year will be much larger than the data collected in a month. However, the difference is because of the number of values, as the growth of keys slows down for big data.

4.1.6 Limitations

The asynchronous design is able to solve problems at different levels for a Map-Reduce application. For example it removes the unnecessary serialization and deserialization required when data is produced and consumed on the same node. It also removes sorting and uses hashing to collect all the values belonging to the same key. However, there exists a problem with applications like Join. We stated in the begining of this chapter, that we have divided applications into two categories. Since, in Join, every value is concatenated with the accumulator, the reasons stated in the previous paragraph do not hold any more. The size of the intermediate state now becomes proportional to the number of values that exist. This causes high memory consumption over the entire lifetime of the reducer until the end when "write" function is invoked for every object. Garbage collection causes the program execution to stop which leads to poor performance for applications like Join. We solve this problem in our synchronous design.

4.2 Synchronous Design

In our synchronous design, we use the original design of Hadoop's reducer, where the reducer waits for the outputs from all the mappers. We merge these outputs from all the mappers to collect all the keys and their values together. Every key and all the associated values will be used in the reduce function. Since all the values of the given key are consumed at once by the reducer, we do not need to have intermediate state for every key. This is how Hadoop normally works. We are, however, different in the techniques used to merge these key value pairs. Our objective of not requiring a sort to take place still holds in the synchronous design. The new programming interface is backward compatible, and hence can be used in the Join application too.

We first give a detailed overview of our design. We then present the implementation details.

4.2.1 Overview - Abstract Design

The design decisions for this work are motivated from the application Join, which happens to be one of the most important applications for analytics. Unlike the asynchronous work which can improve performance of many Map-Reduce applications, this design is specific for Join. Since Join is, and will remain one of the most important applications on Hadoop Map-Reduce, we believe it is worth the efforts to have special support for Join in hadoop. These changes co-exist with the asynchronous design, the programmer chooses the API.

Before going into our design we briefly review the Join application in hadoop which is described in the background section 2.3.2. In the map function, the key on which the equi-join takes place is emitted with the values being the fields in the table which have to be joined. As the framework collects all the keys together on the reducers, a join operation takes place in the reduce function. To reduce the memory consumption in the reduce function, the program exploits Hadoop's Map-Reduce design. The key is appended with a tag value, such that while sorting the key belonging to the primary table comes first, followed by the key from the other table. This design allows the values associated with the primary key to be in memory, while the iterator keeps fetching values associated with this key, belonging to the other table. Since only the value from primary key remains in memory, while the other values are short lived objects, this design gives better performance. This is true because JVM's garbage collection algorithms are optimized for short lived objects.

We are able to solve the following problems in the above stated design.

- First, there is enough redundancy while transmitting information. As the input split of the HDFS file will belong to either the primary table or the foreign key table, tagging every key with the same value leads to more data being shuffled across. It is hard to remove this step, since sorting in the reducer will require the tag information.

- Since Join does not need a combine operation, the key is serialized as many times as the values appear for this key in the input split.
- Sorting is just as expensive in Join as it was in the earlier applications that we discussed.

We use a novel scheme which uses hashing to collect the key value pairs together, while also solving the above problems of redundancy. Instead of serializing the key-value pairs, we construct hashmaps in the mappers for every partition. These partitions are distributed to the reducers. The reducers then collect the same keys in these hashmaps. We explain how this collection happens in the next section. Since the number of mappers tends to be more than the number of reducers, construction of the hashmaps is moved to the mappers, while only merging or collection of keys happens in the reducers.

4.2.2 Implementation Details

In the initialization phase of the mappers, we create in memory files for every partition. We create hashmaps in these files. These hashmaps are agnostic to the particular programming language or environment and can be used anywhere with appropriate handles. This hashmap supports multiple values for a key. All the keys and values emitted from the map function are serialized in this hashmap. As explained in the asynchronous design, these files are also written to the disk at the end of the map phase. The reducers fetch these hashmaps and starts merging them.

At the time of fetching these hashmaps, no intermediate merging is done. Hence, the CPU utilization is expected to be low for the reducers, until all the hashmaps have been fetched. After receiving all the outputs, the iterator picks up the first key from the first hashtable, and passes it to the reducer along with the iterator which will be used to fetch the values. In the reducer, for every fetch of the value in the iterator, this key is looked up in the hashmaps until it is found. After the key has been looked up in the last hashmap, it will return null signalling the end of values for this key. The reduce function will return, and the framework will now pick the next key from the first hash map and repeat the procedure. We then delete the file backing this hashmap, once all the keys from this hashmap are exhausted. We now pick the key which has not been looked up in the next hashmap, and this procedure repeats till we reach the last unlooked up key in the last hashmap. Since the performance of joins in the reduce phase will be affected by the design and implementation of this phase, we do various optimizations to improve the performance. We also show next that in case of Join, we do not need to find unlooked up keys in all hashmaps.

4.2.3 Join specific optimizations

As explained in the background section on Join, the values in the reduce function need to be ordered to reduce the consumption of memory. To have the ordering in values, without having to sort the keys, we use a small signature in our hashmaps. This signature contains the priority allotted to the hashmap. While fetching the hashmaps, the reducer checks its priority, and adds it in the appropriate priority list. In the Map-Reduce join application, the programmer will set "0" as the priority if this mapper is responsible for an input split belonging to the primary table, and set a priority of "1" if the input split belongs to the other table. After all the outputs have been fetched, the reducer will have two lists of hashmaps. The first list will have hashmaps belonging to the primary table, while the second list will have hashmaps belonging to the other table. The iterator picks the first key from the first hashtable, and does a lookup in all the tables following it in both the lists. Since the join operation takes place between the keys from one table to another, we need to iterate for keys in the hashmaps which belong in the first list only. This invariant holds for both the join operations, *i.e.* one-to-many and many-to-many. For a one-to-many join, the keys and values can be directly serialized into buffers like in the asynchronous design with no combiners. This is so because, as the first list of hashmaps will have only one copy of every key, a lookup will not be required. The buffer just needs to support an iterator over it. However, for a many-to-many join, we still need hashmaps in the first list.

We now explain the construction of the hashmap which is based on the original work from MemC3[15]. We then explain our optimizations on top of this hashmap.

4.2.4 Hashmap Construction

Our hashmap uses cuckoo hashing as a hashing technique [14]. It supports the usual GET and PUT operations. It also supports an iterator, which enumerates over all the keys in the hashmap. In cuckoo hashing keys can be hashed to two locations or buckets. On a collision with another key, this other key is displaced to its alternate location. This can cause recursive displacements. Specifically, displacements will repeat until a vacant slot is found or the maximum number of displacements have taken place. If all the keys have not been placed in vacant positions till the maximum number of displacements, the hash table is doubled. However, the space utilization is around 50% which is very low. We use the same techniques from the prior work, where the set associativity of the bucket is set to 4. This improves the space utlization to 95%, with the maximum number of displacement set to 250.

Now we explain the construction of this hashmap. Every hashmap is constituted of two files. Since the keys are of variable length, we store the keys and values seperately from the hash table. The hashtable stores the pointers to the keys. To reduce the memory references while doing a lookup, a small summary(1-byte tag) is stored along with the pointers. This tag value is the least significant byte of the hash value. Since tags can collide we need to compare the keys, to ensure it is the correct key. Hence, a negative lookup makes $8/2^8 = 0.03$ pointer dereferences on average. With the pointer size being 4 bytes, every bucket uses 20 bytes in all. As this fits into the CPU cacheline(usually 64 bytes), on average each lookup requires two cacheline reads. These tags are also used to find the alternate locations for the key. This optimization avoids a fetch of the key to calculate alternate locations.

4.2.4.1 Optimization

Our optimization in the construction of hashmaps is based on the following observation. In the reducer, for every key picked up by the iterator, we do a lookup for the key in all the hashmaps which follow it. Since all the hashmaps are constructed from the same input split size and assuming the data is distributed identically in a cluster, we will be fetching the same bucket from all hashmaps. Since our hashmaps are page aligned(side effect of memory mapping the files), these buckets will fall in the same cacheline. This can cause conflict misses, as the set associativity in CPU caches is limited(usually 16 for L3 cache). If the next key picked up by the iterator belongs to the same bucket, it will need to be fetched from memory, since all the previous cache fetches from the hashmaps would have evicted it from the cache.

To fix the above problem, we make use of the mapper id. Mapper-id's are sequential numbers from 0 onwards. Every bucket obtained after applying the hash to the key is offset by the mapper-id. This ensures that identical keys do not all fall in the same bucket for every hashmap. In the reducer while doing a lookup, this offset is added appropriately, as the mapper-id is part of the hashmap signature.

Chapter 5

Analysis and Evaluation

This chapter presents the experimental evaluation of our work. Since this work presents two different models for Map-Reduce computation, we evaluate Hadoop on WordCount, Pagerank for our asynchronous design, and reduce side Join for our synchronous design. We compare the run time performance of the applications in original Hadoop with our modified Hadoop, while also presenting some interesting observations from these benchmarks.

The analysis present in this work reflects the various ineffeciencies present in Hadoop. We observe high overheads in the task engine in activities such as sorting, serialization and deserialization. As state previously that these problems stem from the stateless nature of Map-Reduce functions. We fix this problem by introducing a new interface which allows map and reduce to have side effects. Our new interface remains simple and intuitive to the programmers. Our evaluation of the benchmarks yields that we gain performance improvement from 1.5 to 2.5 times.

5.1 Experimental Setup

We run our experiments on a cluster of 20 m3.xlarge servers on Amazon EC2. The master node or the resource manager is located on a different machine. We use Hadoop 2.2.0 with Java 7 for our results and analysis. Each server has 4 cores of an Intel 8 core Xeon E5-2670 CPU(2.60 GHz) and 15 GB's of RAM. We use a 128MB block size by default for our experiments.

We have configured Hadoop to use a maximum of two concurrent running tasks on every machine at a time. With YARN, the application master is scheduled on one of the nodes belonging to the cluster. Thus, we get a maximum of 39 slots for execution of mappers and reducers at a given time.

5.2 Workload Generation

For our applications, we obtain our data from the following sources. For Word Count, we use Hadoop's random text writer application to generate data. This application allows use to set the number of mappers per node, and the amount of data written from each mapper. For Pagerank, we use real data collected from crawling the web. This data is made available from ClueWeb09 [27]. This data contains the first 50 million English pages with 450 million outlinks. For the application Join, data is generated from a Zipfian distribution. The size of the record in the small table is x, while the size of the record in the large table is y.

5.3 Word Count

We evaluate word count with the following different configurations.

- In this benchmark we use more data that can be processed by our Hadoop cluster at an instant of time. Since we have 40 slots with a 128MB block size, we can process a maximum of 5 GB of data in parallel. We use 15 GB of input data for the word count program. Hadoop has a configuration parameter named "mapred.reduce.slowstart.completed.maps", which is responsible for scheduling reducers depending upon how many mappers have finished their execution. A

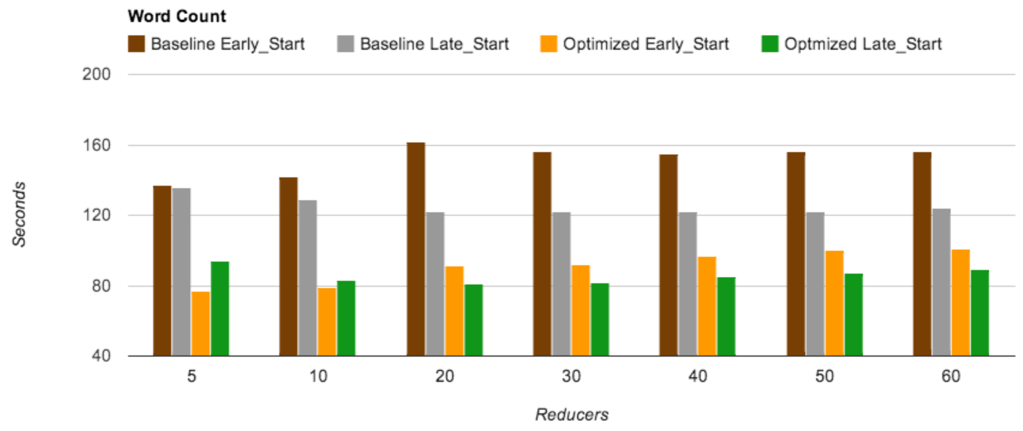

Figure 5.1 : Word-Count performance with increasing number of reducers

value of 0 will start scheduling the reducers immediately, while a value of 1 will wait for all mappers to finish, before the reducers can be scheduled. A value of 0 does not guarantee all the reducers will be scheduled, as that can cause a deadlock. Reducers can not finish before the mappers, and if reducers occupy all the slots, the mappers will starve forever. This value is important because reducers in our modified Hadoop start to process mapper's outputs as soon as they are available.

Let us first compare the results between our modified Hadoop and the original Hadoop for both the values of this configuration parameter. When the slow start is 0, word count on our hadoop cluster is 1.54 to 1.76 times faster, depending upon the number of reducers. We also see that as the number of reducers increases, the time taken to complete the job also increases. This is because, as the number of reducers increase, they occupy a few slots in the cluster, and the new mappers can be scheduled only when the earlier mappers have finished their execution. Word Count is not compute intensive in the reducers, and the loss of parallelism owing to the slots occupied by the reducers causes the map phase to take more time. This results in poor performance despite having a larger number of reducers. However, this is not true when the slow start is 1. As the number of reducers increases, the performance improves as a result of increased I/O and CPU parallelism. This behaviour can be observed in both the baseline version and our optimized Hadoop. Another interesting observation is the performance with 5 reducers. In the case of original Hadoop, the slow start values does not affect the performance. However with our version of Hadoop slow start 0 gives better performance than slow start 1. This is because of our asynchronous computation in the reducers. With slow start 0, by the time the last mapper finishes, the reducers have also progressed enough in their task. In original Hadoop however the actual computation in the reduce function starts only when the last mapper has finished its task.

- In this benchmark we configure the number of mappers and reducers such that they all can fit in the cluster at the same time. We use this benchmark to mitigate any scheduling overheads, and dependency issues which arise when scheduling the mappers and reducers. However, Hadoop will still not schedule all the reducers in the begining. This is because the scheduler is conservative before using all the resources in the cluster. It will launch a few reducers, and then wait for the mappers to make enough progress before scheduling the rest of the reducers. This benchmark consists of 30 mappers and 9 reducers, making a total of 39 slots. Since the application master, which is responsible for managing the application, occupies one slot in the cluster, we are left with 39 slots for the mappers and reducers. Figure 5.2 shows the difference in their performance. Word count on our version of hadoop takes 30% less time to complete.

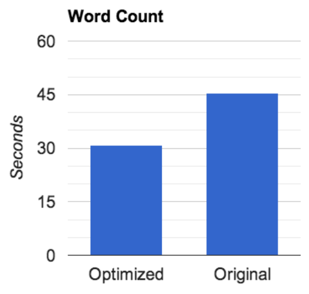

Figure 5.2 : Word count performance with 30 mappers and 9 reducers

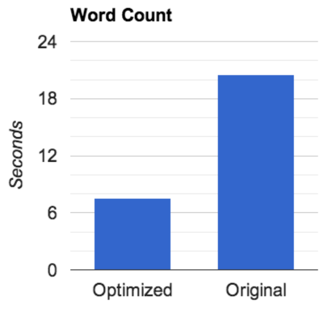

Figure 5.3 : Map Phase Execution time for word-count with 128MB block size

We now present the breakdowns of the word count application. The performance gains that we obtain are mostly or entirely due to the faster execution in the map phase.

Figure 5.3 presents the execution time in the map phase. The map phase in our design outperforms original Hadoop by 2.8 times. The execution time of the mapper depends only on the input split size, which is 128MB in our case. In our design the execution time in the map phase is also dependent upon the number of unique keys. As we introduce the concept of an asynchronous combiner, there exists a live state for

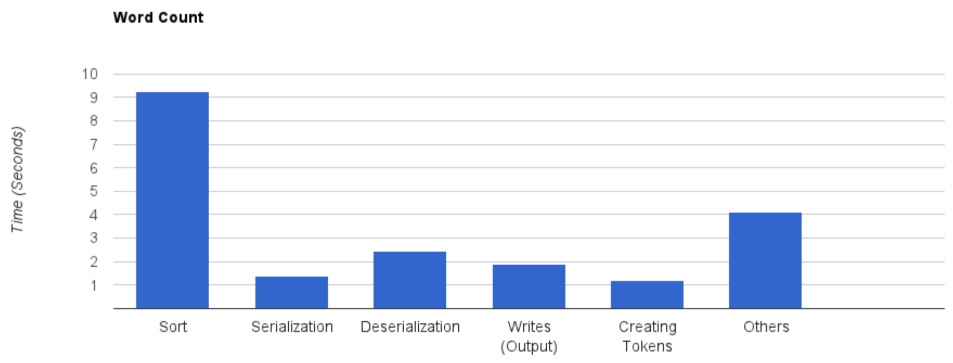

Figure 5.4 : Word-Count performance break down in Map Phase for baseline Hadoop

every key. For pathological cases where the number of unique keys is very large, the garbage collection time can dominate the overall execution time. With a fixed block size, the average number of values per key will determine the number of unique keys.

5.3.1 Application Breakdown

We obtain the break down of the application performance in Hadoop using the standard technique of sampling the program counter.

Figure 5.4 shows the break down in the map phase. Majority of the time in map phase is being spent in sorting. Deserialization takes 13% of the execution time. Deserialization happens when the combiner phase is invocated. This is the phase where key value pairs are fetched from an iterator. As the key value pairs are serialized in memory, the iterator needs to compare every key with the next contiguous key to find where the new key begins. The iterator returns null when a new key is seen. This is done so that the reduce can finish and emit its output, and a new invocation of reduce can be done. The second cost associated with deserialization is the creation of the object from the serialized data. 6.88% of the time spent in map phase is spent in

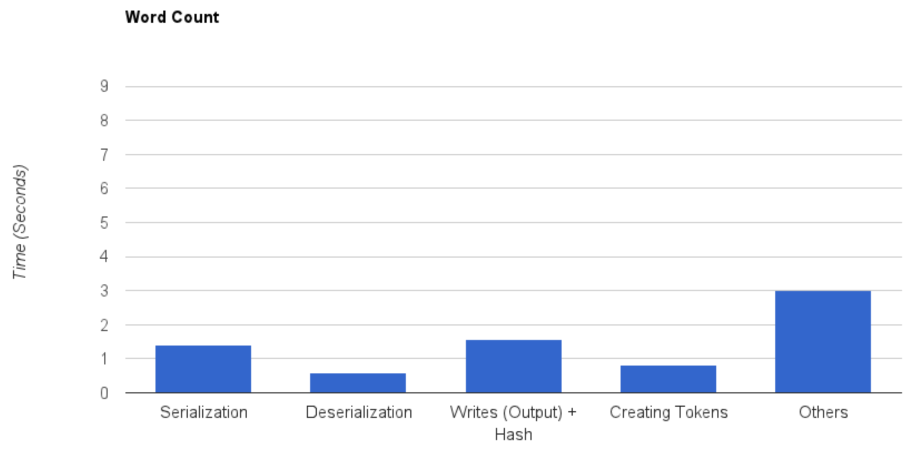

Figure 5.5 : Word-Count performance break down in Map Phase for our modified Hadoop

serializing the String object into a byte stream. This is different from writes, which only account for the time taken when emitting the output of mapper and combiner. Creation of tokens takes 6% of the execution time. Since new objects are created from every line read, StringTokenizer becomes an expensive operation overall.

Figure 5.5 shows the breakdown in our modified Hadoop version. As the execution time is much lower than the original Hadoop, the fraction of time taken by the category other has gone up. Since every write from the map is causing a write into the hash table, we have combined the writes and the cost of hashing into a single entry. 21% of time is spent in writes, and in hash table operations like lookups and insertions. Serialization accounts for 19.8% of the execution time in our modified Hadoop.

5.4 Pagerank

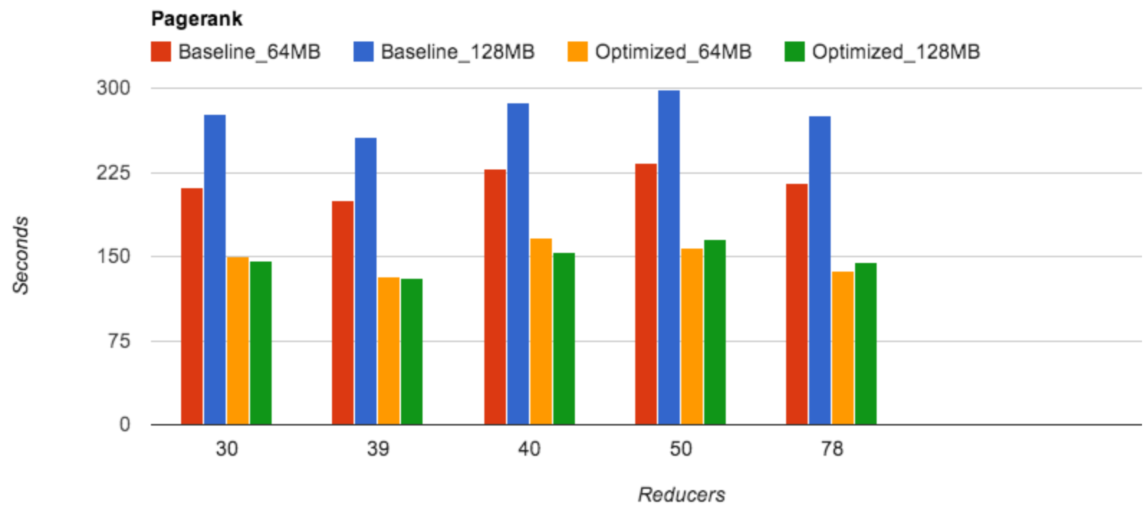

Figure 5.6 : Pagerank performance with increasing reducers and block size

In this benchmark we use real data obtained from crawling the web. The results show that we are 2 times faster as compared with original Hadoop. We present a few interesting observations, along with the conclusions that we can make from the data.

The first interesting observation that we see in this data is that the performance is good for small reducers and large reducers. However, it is not so good for reducers in between. This happens because of poor load balancing between the nodes when computing the reduce phase. Since the maximum available CPU parallelism is 39 slots, the best performance can be obtained only when the reducers are set to a multiple of 39. This will ensure that all the CPU's are involved at all times when computing reduce. It can be clearly seen that if we use 39 reducers, and assuming the data is not skewed, the time taken for the reduce phase will be (t/39). Here we assume that it will take t time for one node to compute reduce for the entire data. If we use 40 reducers instead, then the time taken in the reduce phase will be (t/40 + t/40 = t/20). This effect is what we observe in the benchmark figure 5.6.

5.4.1 Size Invariant Performance

Our benchmarks show that in original Hadoop, the performance is slightly better with a block size of 64MB. While the performance of Hadoop in our version deteriorates a little with 64MB of block. This is because with small block sizes the cost of sorting is less. In other words, sorting two 64MB blocks is faster than sorting a single 128MB block. However, since our design engenders the performance to be linear in the size of the data, the performance is invariant with respect to block size. With 64MB blocks, a larger number of reducers needs to be launched, and the extra overhead of launching these tasks causes the performance to deteriorate a little. However, this does not allow us to use as large a block as possible, since that would cause the intermediate data to overflow on to the disk. With this information, we introduce a simple model for calculating the optimum size of the input split. Suppose the input size of the data is x, the total number of slots in the cluster is y, and the maximum block size for which data does not overflow into disk is z. Then, the optimum input split size can be defined as

$$\text{Optimal input split} = Min(x/y, z)$$

This will give us the optimum input split size, where we use a minimum number of slots, while making the best use of available parallelism. The weakness of this model however is that it assumes all the slots in the cluster will be available to it in a single pass. Since the cluster is usally shared, getting all the slots at the same time is not a good assumption. In this thesis we do not solve this problem, and leave this as an open problem. The above model for deciding the optimal number of mappers is also applicable for finding the optimal number of reducers as well.

We also claim that this model can not work for the original Hadoop design. This is because on one hand selecting a smaller size block is favourable for sorting, however when it comes to I/O bandwidth, a larger block size is preferred. With our design, all the factors require the block size to be as large as possible, limited by the split size that still allows the intermediate data to fit in memory.

If we apply the ideas to the current data we have, a block of 175MB will be required for only one pass by all the mappers. We get a trivial improvement of 6% from the block size of 128MB. We do not run this benchmark on original Hadoop as that will perform much worse than with 128MB blocks.

Although Pagerank performance in original Hadoop is better with 64MB block size, it uses twice as many mappers to achieve that performance. Since a Hadoop cluster is shared amongst many users, more number of mappers can cause longer wait times for the job. This happens because Hadoop guarantees some kind of fairness amongst the jobs. Since there will be contention amongst the tasks from various jobs, having more map tasks can take longer to complete.

5.5 Join

We use an entirely different model of MapReduce computation for Join because of the amount of intermediate state that is created. Our asynchronous reducer can generate upto 6 times more data in the heap before a write is done as compared to the serialized data received from all the mappers. This is because every value is occupying space, whereas in the previous two applications all the values were folded or reduced to a single value which was occupying state.

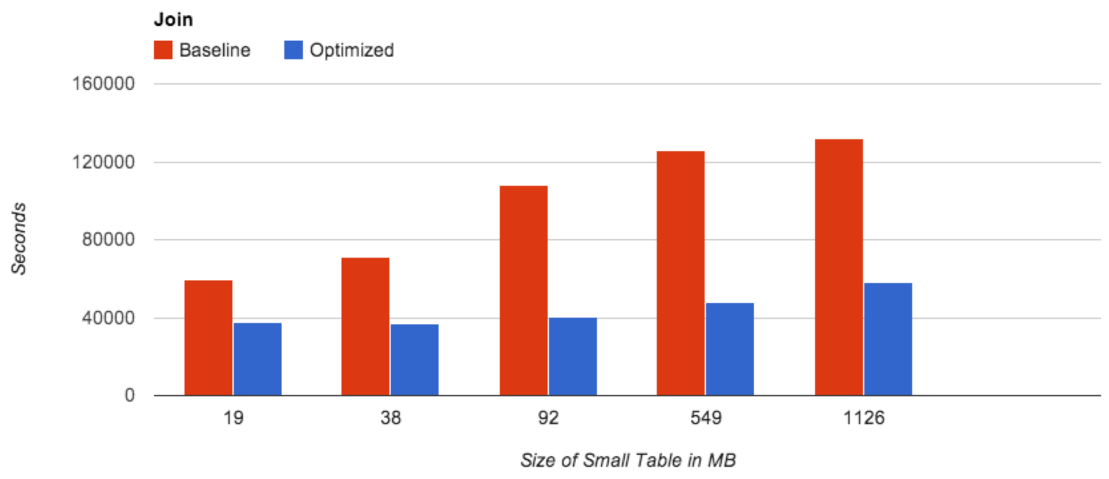

Figure 5.7 : Reduce side Join performance with increasing size of reference table

This benchmark is adapted from the join benchmark described in Pavlo *et al.* [28]. Our benchmark consists of two tables, a ranking table and a uservisits table. The ranking table contains the primary key, and so each key appears only once in this table. Each record is 15 bytes long, where the first 9 bytes consists of the primary key followed by the value. On the other hand the record in the uservisits table is 40 bytes in length. Within the record the value occupies 30 bytes, 9 bytes for the join key, and one byte exists for the delimitter. The value here is a composite of two things, and the join operation filters out the first 15 bytes, and the next 15 bytes are used for the join. The number of times the given key appears in the uservisits table is determined using a zipfian distribution.

In figure 5.7 we show how the performance is affected by keeping the large table as constant, while increasing the size of the small table. In this benchmark we are trying to simulate the effect when the number of unique keys keep growing while the size of the data is same. Thus the average number of values per key is reducing as

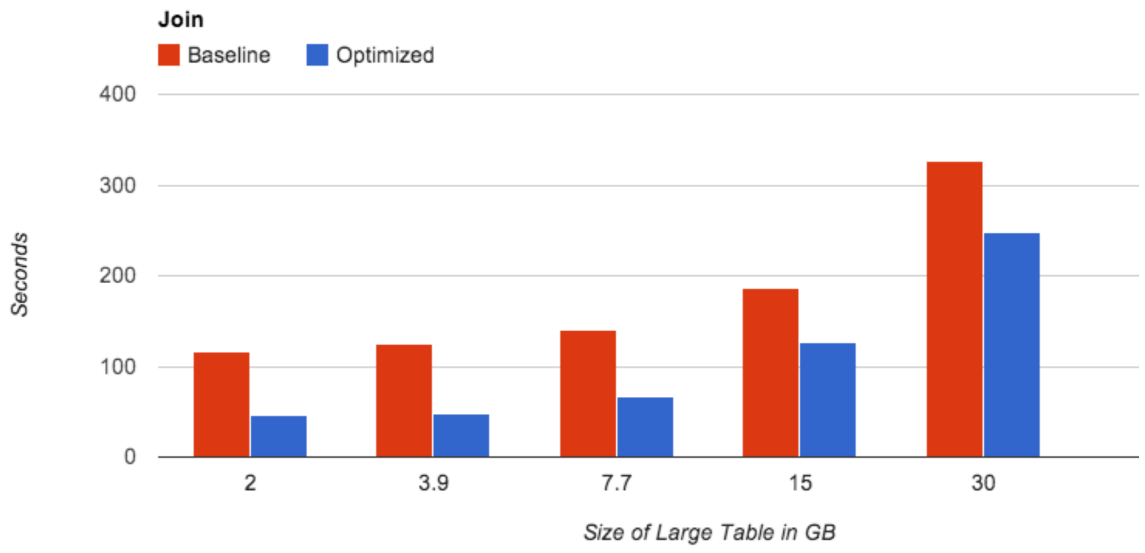

Figure 5.8 : Reduce side Join performance with increasing size of reference table

we keep on increasing the size of small table. Our version of Hadoop outperforms the original Hadoop by a factor of 2.

In figure 5.8 we show how increasing the large table affects the performance. In this benchmark we simulate the effect when the average number of values per key increases. As the large table size increases, the cost of I/O and the amount of work to be done by the reducers increases. Our performance improvement over original Hadoop begins at 2.5 times for the smallest table, and decreases as the table size increases. Nonethelessr, for the largest table size of 30GB, our performance improvement is still 1.5 times.

Since the application join is not itself compute intensive, the computational overheads of the framework constitue the dominant part of the overall execution time. On the reducers, our performance is comparable to what original Hadoop achieves.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presents a more efficient task engine for Map-Reduce. The higher efficiency is achieved by removing or reducing the current overheads of sorting, serialization and deserialization.

This thesis then provides a new interface for Map-Reduce programs which provides better performance. The interface is designed to be simple and intuitive for programmers. This interface removes sorting altogether from the entire pipeline of Map-Reduce programs. Secondly the interface also provides the ability to do asynchronous computation in the reducer. We use popular applications like Word-Count and Pagerank on this interface.

However, for applications like Join this interface is designed to do synchronous computation as well similar to the original Map-Reduce design. This is because Join applications is fundamentally different from applications like Word-Count. We evaluate these applications on this new interface, and with the modified Hadoop task engine. Our evaluations show up to 2.5 times better performance on some of these applications.

Hadoop and Map-Reduce have become the de facto standard for large scale data processing. The work in thesis allows to improve the performance of many Map-Reduce applications on Hadoop. As Hadoop is used on large clusters, improvement of efficiency on such a large scale helps reduce the cost both in terms of time and money.

6.2 Future Work

There are many directions where the work in this thesis can be extended to.

One of the assumptions that this work makes is that there is enough memory for doing the Map-Reduce style computation. On the mappers this can be made certain by deciding the input split size such that the intermediate data does not flow into the disk. However, on the reducers we can never be certain as data can get skewed, and adding more reducers will not be of much help. In such cases where the reducers have more data to process can enter the state where it uses the original design of sorting. Sorting is one of the best known techniques to collect data when it is too big to fit in memory.

Map-Reduce online [11] discussed a possible solution where the mappers output is pushed to the reducers instead of the reducers fetching the output. Pushing the mappers output as soon as the internal buffers are filled, can cause the reducers to do more work, as they need to do the sorting instead of merging. This is not a good idea. Since the mappers tend to be more than the number of reducers, we lose on parallelism if reducers have to do the complete sorting. The paper discusses possible strategies to counter this effect. However, our work is complementary to this design, as there is no need to perform sorting on the key value pairs. In case of asynchronous design, since the order does not matter to the reducers, pushing partial outputs can increase the effective pipelining in the Map-Reducer framework. For our synchronous design, since we use two files for every hashmap, the second file, which contains the data and not the actual hash table can be pushed partially. The hashtable file however will be pushed only at the end of the map phase.

As also stated in our thesis, the Map-Reduce program written in the original programming model can be translated automatically to use our extended programming interface. We believe this is possible because of the mechanical changes needed to be applied to reach our extended interface.

We can also think of a hybrid model of computation here. Since our asynchronous model can bloat the internal JVM heap and cause garbage collections, a hybrid model can solve this problem. If the asynchronous reducer causes memory to cross a particular threshold, the system can automatically move to a state, where it will now wait for all the mappers output to be fetched first, and then process all the values for every key, do a write and free the state or the memory in the heap associated with this key. This is possible because our extended interface is backward compatible.

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[2] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. PhD thesis, EECS Department, University of California, Berkeley, Feb 2014.

[3] D. Borthakur. "facebook has the worlds largest hadoop cluster!": http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html.

[4] http://www.infoq.com/news/2013/12/hadoopusage.

[5] http://www.slideshare.net/cloudera/hw09-data-processing-in-the-enterprise.

[6] http://www.slideshare.net/cloudera/hw09-large-scale-transaction-analysis.

[7] http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun.

[8] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 299–310, New York, NY, USA, 2009. ACM.

[9] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[10] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[11] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[12] Abhishek Verma, Brian Cho, Nicolas Zea, Indranil Gupta, and Roy H. Campbell. Breaking the mapreduce stage barrier. *Cluster Computing*, 16(1):191–206, March 2013.

[13] Dong Yang, Xiang Zhong, Dong Yan, Xusen Yin Fangqin Dai and, Cheng Lian, Zhongliang Zhu, Weihua Jiang, and Gansha Wu. Nativetask: A hadoop compatible framework for high performance. *The First Workshop on Big Data Benchmarks, Performance Optimization, and Emerging hardware.*

[14] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[15] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings*

of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[16] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[18] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[19] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[21] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[22] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 16–52, New York, NY, USA, 1986. ACM.

[23] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[24] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[25] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[26] K. Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. masterâĂŹs thesis, university of edinburgh, 2009.

[27] http://www.lemurproject.org/clueweb09/.

[28] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.