TRANSPARENT IN-MEMORY CACHE FOR HADOOP-MAPREDUCE

by

Venkatesh Nandakumar

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2014 by Venkatesh Nandakumar
Abstract

Transparent in-memory cache for Hadoop-MapReduce

Venkatesh Nandakumar
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2014

Many analytic applications built on Hadoop ecosystem have a propensity to iteratively perform repetitive operations on same input data. To remove the burden of these repetitive operations, new frameworks for MapReduce have been introduced, which make users follow its programming model. We propose a solution to the problem of application rewriting that newer frameworks impose. We re-architected Hadoop core to add in-memory caching & cache-aware task-scheduling.

We set out to match the performance of a state-of-the-art high speed, in-memory MapReduce architecture with caching (Spark). While Spark[1] reimplements the MapReduce paradigm, it comes with a new set of new API’s and abstractions. We maintain the familiar Hadoop framework and API’s, thus complete backward compatibility for any existing Hadoop-based software. This ensures no changes to existing applications code whatsoever. It guarantees no-pain installation over existing deployments while providing 4.5-12X performance improvement. We perform comparable to, and in some cases outperform Spark.
Acknowledgements

I would like to acknowledge my supervisor Dr. Cristiana Amza, for guiding me through my first ever research-oriented work. Dr. Amza is one of the most approachable professors of the university and I really appreciate the time she took to address every one of her student’s concerns and provide valuable feedback for our work. I thank her for her continuous support, without which this work would not have been possible.

I am very grateful to Dr. Eyal de Lara, whose technical inputs and expertise in every step of this work was instrumental. He always had a clear vision of the higher-level picture, which I often missed, and always steered us back in the right direction.

Dr. Ali Hashemi, my colleague and guide, was always there to go into the minutiae details of technical implementation of our work. His insight gave us couple of crucial breakthroughs that I will always be grateful for!

Special thanks to Dr. Bala Venkatesh of Ryerson University, who is the closest thing I’ve had to a family in Toronto.

To my parents, my sister Sambhavi and family, Patti and everybody else who literally own me – Thank you.

I am forever obliged to all my means of financial support: The Department of Electrical & Computer Engineering, Roger’s Scholarship, IBM Markham’s CAS Scholarship, and to my advisor Dr. Cristiana Amza. To BPCL-KR, NTSE and MHRD (Govt. of India) – thank you so much for sponsoring a major part of my formative education.

I’m very indebted to Aakash Nigam and Rohan Bali for all their help in the past two years. To people who’ve made my stay in Toronto memorable - Anmol, Arthi, Cricket guys, Ellen, Gowtham, Hari, Natasha, Nishil, Niharika, Prashanti, Pratishtha, Shailin, Tanisha and Varun - Thanks!

I’d like to thank all my previous shepherds including my really awesome friends from IIT Roorkee, the good folks at Yahoo!, IMG and my teachers & friends from Cochin Refineries School.
# Contents

1 Introduction

2 Background

2.1 Understanding MapReduce
  2.1.1 MapReduce

2.2 Apache Hadoop
  2.2.1 HDFS
  2.2.2 Architecture
  2.2.3 Combiner
  2.2.4 Scheduling and speculative execution
  2.2.5 Spill
  2.2.6 Identity Reducer
  2.2.7 Classes within Hadoop

2.3 Proposal of Benefits
  2.3.1 The Benefits

2.4 Study I – Effect of buffer-cache on a repetitive Hadoop Job
  2.4.1 Introduction
  2.4.2 Experiments
    2.4.2.1 Experiment set A
    2.4.2.2 Experiment set B
  2.4.3 Analysis
    2.4.3.1 Non-cached runs
    2.4.3.2 Cached runs
    2.4.3.3 Profiling
  2.4.4 Results

2.5 Conclusion

iv
# Proposed Architecture & Study II

3.1 Proposed architecture .......................................................... 25
   3.1.1 In-memory cache ......................................................... 25
   3.1.2 Persistency ............................................................... 27
   3.1.3 Cache-aware scheduler ............................................... 28
   3.1.4 Unique identifier ...................................................... 28
   3.1.5 Optional serializer .................................................... 29
   3.1.6 HDFS writeback of Cache ........................................... 29

3.2 Study II – Simulating in-memory key-value store using Ramdisk and SequenceFileFormat ..................................................... 30
   3.2.1 Experiments ............................................................. 30
      3.2.1.1 Read-rate experiment ........................................ 31
      3.2.1.2 Basic SequenceFile experiment ............................ 32
   3.2.2 Modified SequenceFileInputFormat experiments ............... 35

3.3 Conclusion ................................................................. 37

# Implementation

4.1 Persistent Child ............................................................. 39

4.2 In-memory cache and Generic version .................................... 40

4.3 Multiple TaskTracker ...................................................... 41

4.4 Scheduler changes .......................................................... 41

4.5 Other implementation notes ............................................... 42
   4.5.1 Garbage Collector impact ........................................... 42
   4.5.2 HDFSBacked-LRUCache ............................................. 43
   4.5.3 TaskTracker retain of files ........................................ 43
   4.5.4 Serialized versions .................................................. 43
   4.5.5 Optional Cleanup, Setup ........................................... 43
   4.5.6 From high-latency to low-latency ................................ 44

4.6 Hadoop Jobs ................................................................. 44
   4.6.1 WikipediaSearch ..................................................... 44
      4.6.1.1 External tokenizer vs Internal tokenizer ............... 45
   4.6.2 Log analyzer .......................................................... 46

4.7 Stress test & Optimization .................................................. 47

4.8 Conclusion ................................................................. 49
5 Results & Evaluation

5.1 Methodology & configuration ........................................... 50
5.2 Single core, single node WikipediaSearch .......................... 52
  5.2.1 Memory usage & Serialized version performance ............. 55
5.3 Two core, single node experiments .................................... 57
  5.3.1 1GB Input, WikipediaSearch ..................................... 57
  5.3.2 5GB Input ......................................................... 58
    5.3.2.1 WikipediaSearch ........................................... 58
    5.3.2.2 ModLogAnalyze ............................................. 58
5.4 Ten node, two core experiments ...................................... 60
  5.4.1 ModLogAnalyze .................................................. 60
  5.4.2 WikipediaSearch ................................................ 60
5.5 Scaling ......................................................................... 62
5.6 Conclusion ...................................................................... 65

6 Related Work

6.1 Technologies based on Hadoop ......................................... 64
6.2 Recent advancements ..................................................... 65
6.3 Spark & related technologies .......................................... 66
6.4 Map, Reduce output caching mechanisms .......................... 68
6.5 Distributed fine-grained caching ....................................... 69
6.6 Other work ..................................................................... 69
6.7 Conclusion ...................................................................... 70

7 Conclusion & Future Work

7.1 Conclusion ...................................................................... 72
7.2 Future work ..................................................................... 73

Appendices

A

B

  B.1 Scaled SequenceFile experiments with variable split-size ....... 79
    B.1.1 Default-split, (135MB Input File [IF]) experiment .......... 81
    B.1.2 1GB Split, (135MB IF) and 2GB Split, (135MB IF) experiments ........................................ 81
    B.1.3 1GB Split (1350MB IF) Experiment ............................ 82
    B.1.4 1GB and 2GB Split (1350MB IF) No Reduce experiments .. 83

vi
List of Tables

2.1 Non-cached run – subset of collected values ........................................ 19
2.2 Cached run – subset of collected values ............................................. 21

3.1 Speed (MB/s) for read of various file-sizes ........................................... 32
3.2 Time (sec) taken for read of various file-sizes ........................................ 32
3.3 ModGrep – Job completion time (JCT) ................................................... 33
3.4 ModLogAnalyze JCT ............................................................................. 34
3.5 Modified SequenceFileInputFormat with 135GB input, ModLogAnalyze . 36

4.1 Stress test with 136 million objects ....................................................... 48
4.2 Stress test, standalone program (A) and early WikipediaSearch .............. 48

B.1 JCT, Number of Splits/IF with SequenceFile on Hard-drive (HD) or Ramdisk 80
B.2 Properties and runtime of various memcached-based experiments .......... 85
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>MapReduce overview and phases [2]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Hadoop architectural overview and phases [2]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Split-size’s affect on various <em>Job</em> runtime values</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>MapReduce task split histogram (Non-cached) [High res]</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td><em>vmstat</em> and <em>diskstat</em> data over time (Non-Cached)</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td><em>iostat</em> data over time (Non-Cached)</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>MapReduce task split histogram (Cached)</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td><em>vmstat</em> and <em>diskstat</em> data over time (Cached)</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td><em>iostat</em> data over time (Cached)</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>High level overview of proposed architecture</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Detailed proposed architecture for Hadoop</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Graph of <em>ModGrep</em> JCT</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Graph of <em>ModLogAnalyze</em> JCT</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>In-memory cache structure within <em>Mapper</em> class</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Internal-tokenizer vs External-tokenizer and Flow of &lt;K, V&gt; pairs in <em>WikipediaSearch</em></td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Log analyzer Job</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Single Node deployment using at most two cores</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Ten Node deployment using two cores per Light node</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>64MB vs 1GB split input <em>WikipediaSearch</em> JCT</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Single-core, single-node 1GB input <em>WikipediaSearch</em> JCT</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td><em>WikipediaSearch</em> JCT with External Tokenizer</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>Peak memory usage</td>
<td>55</td>
</tr>
<tr>
<td>5.7</td>
<td>Serialized version performance</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>Two-core, Single-node 1GB input <em>WikipediaSearch</em> JCT</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>Two-core, Single-node 5GB input <em>WikipediaSearch</em> JCT</td>
<td>59</td>
</tr>
</tbody>
</table>
5.10 Two-core, Single-node 5GB input ModLogAnalyze JCT  
5.11 Ten-node, Two-core 10GB input ModLogAnalyze JCT  
5.12 Ten-core, Two-core 10GB input WikipediaSearch JCT  
5.13 Scaling factor/core per 512MB of input for WikipediaSearch  

B.1 JCT with HD/Ramdisk as SequenceFile storage; for diff. ‘Split Size (Input File size), Reduce/No Reduce’  

C.1 Java profile graph of a Non-cached run [High-resolution]  
C.2 Profile of a Cached run  
C.3 Profile of the 1024MB Map task reading SequenceFile from Hard-drive  
C.4 Profile of the 1024MB Map task reading SequenceFile from Ramdisk  
C.5 Profile of the 1350MB Map task reading SequenceFile from Hard-drive, devoid of any Reduce/Combiner  
C.6 Profile of the 1350MB Map task reading SequenceFile from Ramdisk, devoid of any Reduce/Combiner
Chapter 1

Introduction

MapReduce\[3\] was introduced in late 2004 by researchers at Google. With the advent of big-data, researchers understood that a single machine cannot possibly serve all data computation/analytic solutions, and that a distributed system was needed to store and process data in parallel. But parallel programming is hard. There was a need to ease communication between nodes, scale easily and make the whole system resistant to failures.

MapReduce paradigms filled this gap by providing an abstract map and reduce programming model while taking care of all the internal details including automated parallelization, input ingest and splits, I/O scheduling, network data transfer, fault tolerance etc.

MapReduce, and it’s corresponding opensource implementation Apache Hadoop\[4\], became popular and was soon being used for large scale data analytics, big-data applications and other major parallel computations that usually had large input data.

Hadoop was adopted by major companies and became mainstay. Yahoo!, Facebook and other large corporations started contributing to Hadoop and it slowly evolved over time.

A number of softwares have been built around Hadoop and are part of it’s ecosystem i.e. all these softwares use Hadoop as its underlying computing engine. These softwares solve various computation problems, which are different from the generic Map and Reduce paradigm. Iterative algorithms, machine learning libraries, graph processing systems and BSP computing\[5\] problems are solved by softwares such as Apache Hive\[6\], Apache Giraph\[7\], Apache Hama\[8\], Apache Mahout\[9\] – all of which harness Hadoop.

Studies have shown that Hadoop, and the analytics tools that run on top of it, ingest the same data over a period (window) of time. Up to 60% of the data ingested by Hadoop Jobs in a production environment is reused\[2\]. Also, Hadoop Jobs tend to be batched in
Chapter 1. Introduction

nature, and interactive analysis of data often reuse the same input and redo a number of similar map operations on the input. For e.g a web server log analysis application usually reads the same logs over multiple runs in a day – since HTTP logs are rotated daily.

Hadoop generally parses the input data to key-value \((<K, V>)\) pairs, which retains only the information relevant to map input. The parsed \(<K, V>\) pairs tend to be much lower in size than the input data. There is even more significant overlap of data-reuse of these \(<K, V>\) pairs among interactive Hadoop Jobs.

Over the past decade, clusters of Hadoop have grown in size, with each machine potentially housing ten-hundred GB’s of memory (RAM). Given the smaller \(<K, V>\) size, the high amount of data-reuse and Hadoop Job interactivity, it makes sense to build a robust caching mechanism – preferably in-memory – for significant performance improvements.

A number of research initiatives pushed towards caching in Hadoop, or in-memory caching of \(<K, V>\) pairs in MapReduce. But only rarely was the caching done correctly. HaLoop[10], for e.g cached the parsed \(<K, V>\) pairs in a different process than the worker processes performing the actual map and reduce. Therefore, over-the-network transfer of \(<K, V>\) pairs was the bottleneck for HaLoop – resulting in only a 1.85x speedup. Twister[11], adopted a pub-sub messaging model for MapReduce and concentrated on iterative algorithms. This had a new programming model, and also introduced new API’s – not favoring adoption.

Spark[1], is the latest in-memory MapReduce architecture with caching which has gained some traction within the community. Spark implemented caching within new in-memory data-structures. With thread-based workers, it was very performant and resulted in significant speedup. But since Spark built the MapReduce paradigm from ground up, it had its own new programming model using Scala[1]. With Spark, Jobs had to explicitly specify the \(<K, V>\) pairs that needed to be cached, and porting effort from Hadoop to Spark’s Scala is costly. To move from Hadoop to Spark, existing code had to be re-written in Scala and in-memory caches had to be defined with the new APIs. The Hadoop workflow of Jobs and pipes had to be rewritten. Certain amount of re-architecting and re-deployment of clusters was also necessary. Also, Spark did not have an ecosystem of softwares as mature as Hadoop.

Our goal, in this thesis, was to maintain the very familiar Hadoop API, but provide similar speedups and performance boosts as observed by Spark. We did not want any changes to application code whatsoever. We ensured that there was no re-deployment of application code, or re-architecting of workflows. At the same time, our goal was that the ecosystem of mature Hadoop softwares (such as Hive, Giraph etc) would be able to
benefit from the major performance boosts from our underlying execution engine. We set out to achieve these goals, which we believe would favor major industry adoption.

We proposed a new architecture that modified Hadoop slightly to add in-memory caching with our custom data-structure. We plugged in a new cache-aware task-scheduler to help schedule the right task to the corresponding previously-cached worker. To ensure that the in-memory cache is in the same process that did the map operations, we adopted a persistent worker process design i.e our worker processes are not killed after Job complete for easier sharing of the in-memory caches.

Our architecture and implementation was made as generic as possible to cater to every possible Hadoop application, so that we have full backward compatibility. No code change is required for any Hadoop application or tools based on Hadoop to take advantage of this new architecture.

Our design hinged on three basic improvements – the parsing improvement, the in-memory improvement and the reuse improvement. The parsing improvement essentially meant that we cached parsed \( <K, V> \) pairs to save on parsing-computation time on later Hadoop Jobs working on the same data. The in-memory improvement – gained by reading the \( <K, V> \) pairs from memory instead of disk. Finally, the reuse improvement means that we had persistent process based workers; we did not have to re-spawn and re-orchestrate task processes – resulting in further time gain. The persistent process design also enabled easier in-memory data-structure sharing as mentioned earlier.

In our evaluation, our new architecture consistently matched Spark’s performance. Our performance was 4.5x-12x faster than unmodified Hadoop on which our implementation was based on. Compared to Spark, our Job completion time (JCT) for all experiments lagged at most by 5-10 seconds - mostly because Hadoop has higher task-scheduling and orchestration requirements. This lag is less than 5% of total unmodified Hadoop execution time. In experiments where the amount of data processed per processing unit (core) is high, our implementation was faster than Spark. We outperformed Spark in those cases by about 28%.

Given that most of the Hadoop Jobs we ran in our experiments were ad-hoc, with no code changes to adjust with the new architecture – our performance boost comes at no cost. We also did not have to explicitly set the data-caches using any new API’s. This meant much quicker industry adoption of this new architecture for immediate benefits and results.

Our basic prototype implementation of the new architecture is preliminary and does not show the full potential of this architecture. Future work, including a refined thread-based implementation, automatic scaling and zero-copying serialization mechanism for
better storage of the cache will be very helpful in realizing all the benefits.

The next Chapter gives a brief introduction to MapReduce, Hadoop and recaps its basic architecture. This chapter also delves into a background study that we conducted to understand the behaviour of Hadoop Jobs with an in-memory file-system. Chapter 3 proposes the new architecture of Hadoop and defines the new features. We then perform a baseline experiment. The 4th Chapter on implementation goes deep into the technical details of the prototype we made. We also talk briefly about two major Hadoop Jobs we used for our evaluation and the stress-test we performed to understand our implementation bottlenecks. Chapter 5, introduces the deployment configurations and evaluation methodology. We then analyze the results of experiments conducted on both single-node and ten-node configurations. This is followed by related work and a final chapter that concludes this thesis and probes into possible future work.
Chapter 2

Background

In this Chapter, we will review some of the overall concepts of MapReduce[3] and related technologies which this work is based on. In the first section, we’ll look into the basics of MapReduce and see how MapReduce and Hadoop[4] evolved over the years.

Apache Hadoop, an opensource implementation of the MapReduce concept is introduced in the next section. We then recap the overall architecture of Hadoop and some of the internal details which are relevant to this thesis. Our contribution in this thesis, the new architecture of Hadoop and other features, heavily references the unmodified Hadoop architecture and its internal classes, therefore a solid understanding of the internal working of Hadoop is crucial.

We then explain in detail an experiment we carried out to mimic an in-memory filesystem for Hadoop using the Linux buffer cache. This experiment lead us to the various possible improvements that we could bring to Hadoop by very slightly re-architecting it. We also understood the performance bottlenecks with the current unmodified Hadoop architecture. This also gave us necessary clues for designing our own in-memory cache for Hadoop.

We also summarize and introduce a list of “benefits” or improvements that we conclude from the study. This will help readers to correlate the experiments and new architecture proposed, with the benefits.

2.1 Understanding MapReduce

MapReduce is a 10-year technology that was laid out in the paper by the same name[3] by researchers at Google. The technique is almost synonymous with distributed computing and large-scale data analytics. MapReduce has redefined the magnitude of data and computation that corporations now deal with on a daily basis.
Chapter 2. Background

Figure 2.1: MapReduce overview and phases [2]

The high level overview presented in following sections up-and-until 2.3 does not go in-depth into the technicalities of MapReduce or Hadoop. It has been written assuming that the reader has previous knowledge in this area – and that sections 2.1.1 to 2.3 would serve as a refresher – which is crucial for understanding the remaining of this thesis.

2.1.1 MapReduce

The notion of MapReduce paradigm is simple:

1. A large number of distributed computing problems could be split and solved with the basic operations map and reduce (similar to Lisp[12]).

2. A large number of the said computing problems or algorithms could be re-written/re-architected to be embarrassingly parallel. Therefore, running the computing problem on $N$ of cores could theoretically (ideally) give a $N$-time speedup.

3. By developing a system which creates an abstraction to solve a generic map-reduce parallel computation problem, it was possible to expose certain high-level API’s to developers who did not have to worry about the internal details that involved data-shuffling, partitioning, network topology etc. This was the basis of MapReduce.

MapReduce reduces complexity of parallel computing, and enables easier pain-free, reliable execution on commodity hardware ensuring high levels of fault-tolerance. The high-level flow diagram is given in Figure 2.1. The birds-eye view of working of a MapReduce implementation is as follows:
The input (usually files) are read from a distributed file-system (DFS) such as HDFS[13] or the GFS[14]. Distributed file-systems generally have a master-slave(s) configuration which may piggyback on the same master-slave(s) as that of MapReduce, e.g. as in the case of Apache Hadoop. The input is split into a configurable split.size (commonly 64MB) and each of the splits is processed by a Map task.

The Map task reads the input split, parses it and produces a list of input key-value (<K, V>) pairs.

The Map task performs the map operation on the input <K, V> pairs and produces a list of <K, V> pairs as the output. These <K, V> pairs produced by each Mapper are called intermediate data. It is usually buffered in memory, but may be written (spilled) to disk if it exceeds a configurable value. The Map tasks may or may not be computationally intensive.

The output <K, V> pairs of all Map tasks are partitioned using a customizable Partitioner which produces N-partitions. The Partitioner usually operates using only the Key. N is usually a multiple of the configured number of Reduce tasks.

Each of the uniquely partitioned <K, V> pairs are sent to the same particular Reduce task.

To achieve the last two tasks seamlessly, huge amount of shuffling of intermediate data takes place, and there occurs a network intensive transfer of data from Map tasks to the Reduce tasks.

The Reduce task collects all the <K, V> pairs identified by a partition (usually the key or hash of key), and processes the collection of Values as specified by the reduce operation. Again, this may be compute intensive.

The Reduce task writes back the output to the DFS. If the (configurable) number of Reduce tasks to be spawned is set to 1, then we have a monolithic file as output.

2.2 Apache Hadoop

Apache Hadoop is an implementation of the MapReduce paradigm, which came into fruition around early 2007. The project is open-sourced with primary contributors being Yahoo! and Facebook. The overall architectural diagram is in Figure [2.2].
Chapter 2. Background

2.2.1 HDFS

Hadoop Distributed File System, or HDFS, is an alternate implementation of the Google File System (GFS) as specified in the original paper[13]. HDFS is distributed, scalable and reliable. It replicates data across its slaves, usually by a replicating factor of 3, to achieve fault-tolerance. Files are usually split into 64MB chunks (controlled by the dfs.block.size parameter) before storing into HDFS. This customarily is equal to the input size per Map task which is also 64MB (configurable using mapred.max.split.size). HDFS stores its metadata in the Hadoop’s NameNode[3], which acts in a similar fashion to GFS’s master; the actual data splits are stored in DataNode’s, which correspond to GFS’s ChunkServers.

Now, with certain amount of co-ordination between the NameNode and other Hadoop components, it is possible to schedule the individual Map or Reduce tasks such that there is maximum amount of data-locality. Therefore, the compute aspect of MapReduce is moved to the slave which contains the data locally.

2.2.2 Architecture

The architectural diagram is shown in Figure [2.2]. A typical Hadoop deployment consists of a single master, and multiple slaves. The master is the only possible Single Point of Failure (SPOF). Both MapReduce and HDFS components run side-by-side on the same set of masters and slaves.

The HDFS components include NameNode and DataNode, as mentioned in the previous section. The SPOF in Hadoop, namely the NameNode, has been addressed in
the later versions of Hadoop with a High-Availability (HA) NameNode using a journal manager.

The high-level MapReduce components include the JobTracker and the TaskTracker. The JobTracker is to whom the JobClient submits MapReduce Jobs[15]. The JobTracker creates Map, Reduce task splits and schedules them to TaskTrackers running on individual slave machines. The scheduling is done keeping in mind the data-locality of the splits as exposed by HDFS.

The TaskTracker spawns new Java Virtual Machine (JVM) Child processes for each of the tasks assigned by the JobTracker.

Each of the Map tasks produces intermediate output, which is then partitioned and shuffled to Reduce tasks. The partitioning and shuffling of data functions proceed in the same way as discussed in the previous section.

Hadoop, similar to generic MapReduce, is network-intensive during the shuffle phase. The TaskTracker and JobTracker communicate with each other using hearbeats, which seemingly resemble a polling mechanism. The Child processes spawned by the TaskTracker also communicate back to it via a Remote Procedure Call (RPC).

Hadoop is termed a “high-latency system”. i.e. run-time of a whole Hadoop Job is usually on the order of minutes. The same applies to the time required to create multiple Child processes and the time required for RPC/heartbeat network polling between JobTracker, TaskTracker and Child; There is also a rather network intensive and hence time-consuming shuffle phase. Furthermore, ±10-20 second variation in Job execution time is not unusual.

With a configurable boolean parameter mapred.job.reuse.jvm.num.tasks, it is possible to reuse the same Child processes over multiple Map/Reduce tasks. This lowers Job execution time when each of the TaskTracker is responsible for executing multiple Map/Reduce tasks since it does not have to create a new Child process for every task. It is, however, important to note that the Child processes are not reused across Jobs. i.e. the Child processes are not persistent and are killed at the end of a Job.

2.2.3 Combiner

Hadoop provides an option to supply a Combiner class, to supplement the Reducer function. This is done purely for efficiency reasons. The Combiner function is usually very similar to the Reducer, and is executed on the subset of intermediate pairs generated by a Map task. The function is executed within the context of the Map Child process, thereby decreasing the number of reduce operations to be performed by the Reduce task.
later on.

The Combiner acts on in-memory intermediate pairs, making it fast. The intermediate key-value pairs are separated in-memory identified by the key (Partitioner). They are collected into separate lists, and go through the combiner function before being flushed to the output.

2.2.4 Scheduling and speculative execution

Scheduling of multiple Jobs in Hadoop is governed by the algorithm that is chosen. The default algorithm is First-In First-Out (FIFO), while other scheduling algorithms can be plugged-in using Hadoop’s interface. Other popular schedulers that can be plugged in are for e.g, Fair Scheduler, Capacity Scheduler. These schedulers also determine the methodology by which individual tasks are assigned to the TaskTrackers. The TaskTracker trigger the scheduling algorithm when it sends a heartbeat to the JobTracker and indicates that it has a free Map/Reduce task slot available.

Hadoop also performs Speculative execution, where it assigns the final few incomplete tasks over multiple TaskTrackers. It does so to work-around those slaves which have been bogged down by other unexpected/unavoidable slowdowns. This is similar to MapReduce’s straggler mechanism, as described in the original implementation[3].

2.2.5 Spill

The intermediate key-value pairs are mostly buffered and shuffled in-memory, but if this exceeds a threshold value (io.sort.spill.percent), then it is spilled (written) to disk. Map spills are stored to local slave disk. The spilled data is then shuffled/sorted and written back to the local slave disk as Map output.

The Map output is fetched by the Reducer using normal HTTP-get. It is only the final Job or MapReduce output that is saved back to HDFS and replicated.

2.2.6 Identity Reducer

Hadoop provides an option of Identity Reducer which performs no reduce operation, writing all partitioned and shuffled input $<K, V>$ pairs as is to output. It is also possible to avoid partitioning and shuffling of $<K, V>$ pairs altogether by setting the number of Reduce tasks to 0.
2.2.7 Classes within Hadoop

To understand the contributions of this thesis, it is imperative to know some of the internal Classes of Hadoop, and their relationships. A basic overview of the classes are presented below. The explanation of sample Hadoop Jobs in the coming sections (4.6) further elucidates the \( <K, V> \) flow between the classes. We mostly concentrate on the Map-related classes of Hadoop.

- Each of the Map tasks take in InputSplit[16] as the input, the most common extension of which is the FileSplit – that logically represent the individual file splits read per Mapper.

- The Mapper is configured with an InputFormat[17, 18, 19], which specify the classes of the \( <K, V> \) pairs that are provided as input to the Map tasks. The intermediate map output \( <K, V> \) pairs are determined by the extendable Mapper class.

- The InputFormat is able to generate the input \( <K, V> \) pairs by parsing the Input-Split’s (or FileSplits) using an instance of RecordReader, which is specified by the InputFormat.

- The Reducer, which takes in shuffled, partitioned and possibly combined intermediate \( <K, V> \) pairs as input, runs within the context of the Reduce task and produces output \( <K, V> \) pairs as defined by the OutputFormat.

- SequenceFile is the file-format in which intermediate Map outputs are stored in disk. Corresponding classes SequenceFileInputFormat and SequenceFileOutputFormat are able to read and write SequenceFile’s. The SequenceFile’s or Map outputs are again, only stored to local slave disk and not to HDFS.

- The Partitioner class is responsible for partitioning and sorting intermediate \( <K, V> \) pairs. The default Partitioner used in Hadoop is the HashPartitioner class, which uses the key Object’s hashcode function to create unique partition buckets. The number of buckets is less than or equal to the number of Reduce tasks specified.

- The JobTracker, TaskTracker class correspond to the individual architectural daemons which are shown in Figure 2.2. The Child class embodies the process, spawned by the TaskTracker, within the slave.

- The Child runs either the Mapper or a Reducer class instance. A MapTask or ReduceTask object, is the primary representation of the task to be executed on the Child. The MapTask/ReduceTask objects are transmitted over the network when assigning duties to TaskTracker or to the Child.
• *JobClient* represents the clients from Figure 2.2. It submits a *Job* to the *JobTracker*. The *JobTracker* creates an *JobInProgress* object for the submitted *Job*. This new *JobInProgress* object initializes with *MapTask-split* and *ReduceTask-split* objects.

• *JobQueueTaskScheduler* is the default FIFO scheduler for Hadoop, and is triggered every time a *TaskTracker* sends a *heartbeat* requesting a new *Task*.

• The *TaskTracker* has a *JVMManager* class to manage the individual JVMs (*Child*); and a *TaskInProgress* class to manage all currently running *MapTask* and *ReduceTask* on its node.

### 2.3 Proposal of Benefits

Through a number of experiments and background study, explained in detail in the following sections, we understood the core areas as to where the performance of Hadoop lagged. Some of the bottlenecks were almost immediately obvious right from the start, while some other areas of possible improvement could only be clearly identified through our many experiments. Many of the possible areas of improvement could not be properly quantified, because of a lack of overall profiling tool for Hadoop.

We readily recognized that a slight re-architecture of Hadoop solving these bottlenecks could bring about great performance boosts, while still being possible to maintain backward compatibility and Hadoop API’s – as a result we finally were able to design and implement the transparent in-memory cache-architecture with persistent process-workers as we’ll see later in Section 3.1.

The following subsection has been introduced earlier in the thesis, for readers to correlate the experiment study in the following sections with the proposed benefits. The proposed architecture and benefits consists of a lot of *hindsight* work. It was proposed, repeatedly modified and finalized upon after multiple iterations of our experiments. A lot of the proposed architecture was re-evaluated and modified well into the implementation phase (Section 4), with increased understanding of the underlying Hadoop code.

#### 2.3.1 The Benefits

A number of slight internal changes to the Hadoop architecture could possibly bring a large performance improvement to repeat Hadoop *Jobs*, or *Jobs* that require a large number of iterations, or just *Map* tasks that work on the same *InputSplit* as previous *Maps*. Some of the possible *improvements* can be quantified as below:
• **Parsing Benefit** – Most ML (machine learning) algorithms, iterative computations, interactive queries and flow-based Hadoop Jobs ingest and parse a lot of the same input files to produce input \(<K, V>\) pairs. This could be easily avoided by possibly storing the parsed output to disk as SequenceFiles or storing the parsed output using other storage mechanism such as in-memory cache. This benefit, of avoiding needless repeat parsing of InputSplit by the RecordReader is called the Parsing Benefit.

We’ll see from a later experiment that up-to 60% of the computation time spent on a Hadoop Job Map task is for reading and parsing input data. (Section 2.4.3.3)

• **In-Memory Benefit** – Memory reads are an order of magnitude faster than disk based reads. Approximate speed/time benefits have further been showcased and experimented in section 3.2.1.1. Hadoop could possibly lower the input/output (IO) wait of Map tasks by using ramdisk-based HDFS. We could also, couple the Parsing benefit with in-memory based reads to have even lower read times. Reading back Java Objects in a JVM of previously collected \(<K, V>\) pairs would be the fastest possible way. This gain of time by lowering IO wait by reading data from Memory/RAM is what we term as the In-Memory benefit.

In another later experiment (Section 3.2.1.1), we’ll observe that memory-read of file-sizes comparable to that of Hadoop-split input is faster than disk-reads by upto 10x.

• **Reuse of Containers/Child Process Benefit** – Hadoop tasks by default run on separately spawned Child JVM process. The `mapred.job.reuse.jvm.num.tasks` configuration parameter allows Jobs to reuse the already spawned JVM processes (Section 2.2.2). But across multiple Jobs, the Child processes are non-persistent. i.e all Child processes are killed off at the end of a Job.

This could be avoided, saving Child process spawn time for multiple tasks. This has been termed the Reuse benefit. For e.g, a persistent Child process, coupled with in-memory Java Objects (In-memory benefit) containing pre-parsed input \(<K, V>\) pairs (Parsing benefit) would mean monumental decrease in overall Map execution time. This is, of course, dependent on the computational intensity of the actual map function. Having persistent process-workers also enables easier-sharing of in-memory data-structure across different Jobs – since the cache is retained in-memory and not killed once the Job completes.

In our experiments, we found that the orchestration tasks of Hadoop, which includes Child spawn and task assignment, amounted to upto 30% of Job run-time (Section 2.4.3.3).
Chapter 2. Background

Figure 2.3: Split-size’s affect on various Job runtime values

- **Heartbeat/Orchestration Benefit** – Hadoop was developed at a time when distributed event-based systems and programming were not mainstream. Hence it employed a simpler polling mechanism where TaskTrackers periodically notifies JobTracker of its status. The Child JVM process also communicates with the parent TaskTracker over RPC. This gets slow when scaled to a number of tasks, especially when there are numerous TaskTrackers per node. Scheduling and assigning of tasks/splits also gets bogged down. Any better replacement of this heartbeat mechanism would likely show improved performance, which is what we term as the Orchestration benefit. Replacing Hadoop’s code with an event-driven mechanism for e.g., would be a possible solution. But this will be a huge undertaking for a codebase as big as Hadoop.

- **Process vs Thread Benefit** – Instead of individual Child process, if it is possible to spawn threads instead, we could gain an edge given that thread spawning is cheaper. We call this the Thread benefit. Threads, also will scale up better than processes.

2.4 Study I – Effect of buffer-cache on a repetitive Hadoop Job

Taking inspiration from Spark[1], we were interested in a in-memory cache architecture for Hadoop. But to get an early idea of efficacy of such an architecture, we conducted an
experiment that read input data from Linux buffer cache – simulating a simple in-memory data-read.

Ingested data was already shared by over 60% in production systems[2], and therefore in-memory cache should definitely be beneficial overall.

By the end of this experiment, we were convinced of the following:

i) An in-memory cache architecture would indeed be helpful and give significant performance boosts.

ii) Reading input-data from disk and parsing them to $<K, V>$ pairs for map input is significant ($\sim$60%), if this can be cached in a new architecture, then we would save that much computation time.

iii) User-code computation time is relatively low (less than 30%) even for (what was regarded as) a computationally intensive Job.

iv) Lot of time was being spent on Hadoop orchestration, coordination and task-spawning, assignment.

This helped us propose the improvements (Section 2.3) to be implemented in the new architecture (Section 3.1).

### 2.4.1 Introduction

We wanted to get an idea of how well the disk input data, after having been read and cached over a Hadoop Job would affect the next iteration of the same Hadoop Job run, given that the input data fits within the RAM.

Since disk reads are slower than cache (Memory) reads, we expected the time taken to be much lower. The more pertinent questions were:

i) How was the map time, shuffle time, reduce time distributed in the MapReduce[3] (MR) Jobs?

ii) The difference between time taken for Hard-drive input read vs RAM-cached input read for MapReduce tasks?

iii) The time taken for parsing/ converting to internal format ($RecordReader$[20] - $<K, V>$ pairs) suitable for Hadoop Map input.

iv) User-code time (Regular Expression (regex)[21] matches), etc.

### 2.4.2 Experiments

There were two set of experiments run. The basic setup and run of the experiments are as follows:
2.4.2.1 Experiment set A

In a single compute-node setup of MR2[22], we ran a modified Grep[23, 24] Job in succession. The Grep Job searched for the count of the number of instances of the word “output”. The input source of data was a 10GB Wikipedia dump, split into ten – 1GB files. They were already loaded onto HDFS.

The single compute-node was an Intel Xeon, SunFire X2250 8 core machine with 16GB RAM running Linux. While idling, and with Hadoop running, the RAM consumption was usually less than 1GB. This ensured that storing the entirety of 10GB wiki-dump was possible within the cache. Another machine of a similar configuration was used as the Resource Manager (of YARN), Applications Manager and other house-keeping tasks associated with the YARN-MR2 framework.

Grep[23, 24] was modified to discard the sort operation. Since, we were searching only for the whole word (“output”) rather than a regex-match, sort operation was unnecessary. Also, reducing the number of operations within the Grep Job lowered the complexity of the cache-study we were supposed to conduct. The number of Reduce tasks was limited at 1.

In the non-cached run of the experiment, the cache of the linux machine was dumped before the start of the experiment. Then, the Grep Job was triggered. This ensured that the data was read purely from the disk. This was repeated in 5 separate runs of the non-cached experiment.

The cached run of the experiment was executed immediately after the all the non-cached runs were completed. But here, the cache sync/dump was not performed. Therefore, it was expected that all the input data be read from the cache itself.

Various tools were used to observe and plot various parameters of the machine during the course of these experiments. vmstat, /proc/diskstats, and iostat were the tools used and they often overlapped with respect to the parameters being monitored. Additionally, Hadoop history (.jhist) files were parsed and additional data was collected from them. The MR JobViewer was also modified to spill out more relevant data. Various graphs were plotted from the data collected, as explained below.

The 1st graph plotted, from values collected using /proc/diskstats and vmstat, mapped:

i) The occupied cache over time of MR Job run.

ii) The disk Sector Read (MB) and Write (MB) values over time. [Cumulative]

iii) The disk Read (MB) and Write (MB) values. Non-cumulative, over the last 1 second; plotted using the secondary y-axis.
iv) They also marked the launch time of the MR Job, the actual start time of the Map tasks, the start of the Reduce task, the end of Map task, the end of Shuffle and collect (of the Reduce task), and the final finish time of Reduce (and usually of the whole MR Job).

The 2nd graph plotted, from values collected using iostat, mapped:

i) The number of read and write operations over time, per second. Non-cumulative.

ii) The disk Read (MB) and Write (MB) values. Non-Cumulative, over the last 1 second; plotted using the secondary y-axis.

iii) It also plots the same important time markers as the previous graph (launch time, MR start time, etc).

The 3rd graph plotted, from values collected from the .jhist files and parsed logs, mapped:

i) A histogram of individual split of Map and Reduce tasks over time. The concurrent Map tasks plot over time.

ii) Split of the single Reduce task showing the Shuffle and the Reduce.

iii) It also plots the same important time markers as the previous graph (launch time, MR start time, etc).

2.4.2.2 Experiment set B

This almost identical experiment, differed in only one way: it only collected profiling data of the Map tasks. Map tasks 50-55 (out of the possible 0-79) were profiled using HPROF[25]. Since a perfect timed profiling could not be done[26], we profiled with the next best alternative – using a sampling methodology with 1ms sampling gap.

Call graphs were generated using the profiled data in both cached/non-cached cases[27]. Results were analyzed and using the profiling data, we could see where the maximum time was spent within the MR Jobs for the experiment runs.

The run times of these profiled Jobs were higher than when profiling was turned off.

2.4.3 Analysis

The input data was split into 128MB chunks, and were processed each by a Map task. 6 Map tasks are spawned in the beginning and process their individual splits. A total of 80 Map tasks and 1 Reduce task completes the MR Job.
2.4.3.1 Non-cached runs

In the non-cached runs, we made the following observations:

i) The total run-time was around 194 seconds [±5 seconds]. Figure [2.5]

ii) There is a 4 second delay between the actual Job submit time and the spawning of Map tasks. This is what we call the setup time, which is quite high. Figure [2.5]

iii) The Reduce task is spawned at around the 54 second mark. Figure [2.5]

iv) The amount data read from disk is higher than the amount of bytes written to disk. Figure [2.5]

v) The disk read, cumulatively reaches the 10GB mark over time, and is almost the exact same plot as the cache-size over time. Figure [2.5]

vi) Once the Map tasks complete, the disk read stops abruptly, and the cache-size comes to a standstill. Figure [2.5]

vii) The Reduce task finishes ≤ 1 second after the Map task finishes. The Shuffle finishes almost immediately with the finish of the Map Task. Figure [2.5], Figure [2.4]

viii) The 2nd iostat graph corroborates the values we see in the 1st (diskstat, vmstat) graph. Figure [2.6]

ix) In the 3rd graph, we notice the same setup-time before the spawn of Map tasks. Figure [2.4]

x) Initial Map tasks take about ∼16 seconds, and the later Map tasks take about ∼6 seconds. Averages around 7 seconds per Map task. Figure [2.4]

xi) The system spawns 6 Map tasks concurrently. Once a Map finishes, another Map is spawned. Over time, these Map tasks get staggered – and at any given point of time, the number of running Map tasks is usually lower than 6. As a result, the later Map tasks finish much faster. There is no caching between the disk reads because the split of input data is clean and does not overlap. Figure [2.4]

xii) The Reduce/Shuffle task is pretty much waiting the whole time for the Mappers to finish – the final reduce takes less than a second to finish. Figure [2.4]

xiii) All the 5 runs, produced more or less the same graphs. Variation of ±5 seconds in total runtime for a high-latency system as this one is not too much a deviation. Figure [2.4], Figure [2.5], Figure [2.6]

xiv) CPU time for a total Map tasks is around 208 seconds. The total time for Map tasks (including IO) is about 871 seconds. So approximately 660 seconds of IO
Table 2.1: Non-cached run – subset of collected values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Map Time</td>
<td>208.44s</td>
</tr>
<tr>
<td>CPU Reduce Time</td>
<td>2.66s</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>211.1s</td>
</tr>
<tr>
<td>HDFS Data read size</td>
<td>10737712480 bytes</td>
</tr>
<tr>
<td>Total Map Time</td>
<td>871.11s</td>
</tr>
<tr>
<td>Total Reduce Time</td>
<td>149.807s</td>
</tr>
</tbody>
</table>

wait was involved. The total time of 871 seconds is higher than the runtime of 194 seconds, since there are 6 concurrent Mapper tasks. Table [2.1]

2.4.3.2 Cached runs

In the Cached runs, we made the following observations:

i) The total run-time was around 119 seconds [±5 seconds]. Figure [2.8]

ii) Similar 4-second Setup time. Figure [2.8]

iii) The Reduce task is spawned at around the 38 second mark. Figure [2.8]

iv) The amount of bytes read from disk is lower than the disk write. Figure [2.8], Figure [2.9]

v) The Cache-size is stable over time, cumulative disk write is around 24MB at the end of the run. Actual disk read is negligible. Figure [2.8], Figure [2.9]

vi) The Reduce task finishes ≤ 1 second after the Map task finishes. The Shuffle finishes almost immediately with the finish of the Map task. Figure [2.8], Figure [2.7]

vii) The 2nd iostat graph corroborates the values we see in the 1st (diskstat, vmstat) graph. Figure [2.9]

viii) Initial Map tasks take about ∼7 seconds, and the later Map tasks take about ∼4 seconds. Averages around 6 seconds per Map task. Figure [2.7]

ix) The same staggering effect that leads to smaller Map tasks execution time as the Job elapses is noticed. Figure [2.7]

x) The Reduce/Shuffle task is pretty much waiting the whole time for the Mappers to finish – the final reduce takes less than a second to finish. Figure [2.7]

xi) CPU time for a total Map tasks is around 230 seconds (almost the same as the Non-cached run). The total time for Map tasks (including IO) is about 496 seconds. So approximately 266 seconds of IO wait was involved. So, the IO read time was $\frac{2}{5}$th
Chapter 2. Background

Figure 2.4: MapReduce task split histogram (Non-cached) [High res]

Figure 2.5: `vmstat` and `diskstat` data over time (Non-Cached)
Chapter 2. Background

Figure 2.6: iostat data over time (Non-Cached)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Map Time</td>
<td>230.06s</td>
</tr>
<tr>
<td>CPU Reduce Time</td>
<td>2.27s</td>
</tr>
<tr>
<td>Total CPU Time</td>
<td>232.33s</td>
</tr>
<tr>
<td>HDFS Data read size</td>
<td>10737712480 bytes</td>
</tr>
<tr>
<td>Total Map Time</td>
<td>496.72s</td>
</tr>
<tr>
<td>Total Reduce Time</td>
<td>91.169s</td>
</tr>
</tbody>
</table>

Table 2.2: Cached run – subset of collected values

of the non-cached version. Here again, the system calls for IO-read were being issued, and RecordReader\[20\] parsing was taking place at some point, so the total benefit in read-access does not exactly correlate with the disk::memory read speeds. Table [2.2] xii)Cached runs was only taking advantage of the in-memory benefit (2.3.1).

2.4.3.3 Profiling

We found that about half of the experiment runs gave readable and properly connected call graphs, because we were using the sampling technique for profiling. From them, we were able to deduce the following results:

i) For non-cached run: About 55% of the time was spent on sun.nio.ch.EPollArrayWrapper.epollWait, which was called by org.apache.hadoop.hdfs.DFSInputStream.read (58%, cumulative), and which was in-turn called by org.apache.hadoop.mapreduce.task.MapContextImpl.nextKeyValue (61%, cumulative). So, the maximum amount of wait-time for a non-cached run is spent on disk-read and parsing operations. Appendix [C.1]
Figure 2.7: MapReduce task split histogram (Cached)

Figure 2.8: *vmstat* and *diskstat* data over time (Cached)
ii) For cached run: Less than 2% of time was spent on `sun.nio.ch.EPollArrayWrapper.epollWait`, and less than 14% (cumulative) for `org.apache.hadoop.hdfs.DFSInputStream.read`. `org.apache.hadoop.mapreduce.task.MapContextImpl.nextKeyValue` takes about 23% (cumulative) of total execution time. In spite of the very low poll-wait time, the total read still is substantial - because of various checksumming and other readStrategies that we see in the call-graph (Appendix C.2). So the cached-run time is approx. 2/5th of the non-cached run – similar to what we saw in the CPU/total-execution time ratio previously. Appendix [C.2]

iii) In the cached-run, the total time for pattern-match by the regex-matcher is about 7.5%. The total percent time for regex-mapper being used for the Grep Job is about 22%. Appendix [C.2]

iv) In the non-cached run, the total time for pattern-match by the regex-matcher is about 1.28%. The total percent time for regex-mapper is about 6.58%. 6.58% for the non-cached map-time is equal (in seconds) to the 22% of the cached-run map-time – going to say that the CPU compute time is the same in both cases. Appendix [C.1]

v) The values observed (for the well-connected graphs) were more or less within ±10% error range. This could be attributed due to the fact that sampling was used as the methodology for profiling the tasks. Appendix [C.1], Figure [C.2]
2.4.4 Results

- Compute time for a cached and non-cached MapReduce Job is more or less the same. What it does differ in, is the amount of time taken for reading input data. From these experiments, we see a 2.5X improvement in the time taken to read input data for a cached-run.

- Individual Map tasks stagger over time and therefore take much lower time to execute in the later part of the run.

- User-code and regex matches time taken was quite low, about 20% for cached run and 7% for a non-cached run, amounting to about 15 sec of total time in each.

2.5 Conclusion

We went through an overview of MapReduce and Hadoop, and explained to some detail the classes and internal architecture of Hadoop.

Our early experiment running a repeat Hadoop Job with Linux buffer cache enabled proves that in-memory caching would be at-least 2.5x faster than a disk-based Job. This experiment further gave us an understanding of how an in-memory cached architecture of Hadoop would benefit from the parsing benefit, the in-memory benefit and reuse (of workers) benefit.

We use the knowledge we learned from above, to propose our new architecture in Chapter 3. This new architecture should take advantage of all the possible improvements that we learnt above, while also ensuring that backward compatibility is met – aligning with the goals of our work.
Chapter 3

Proposed Architecture & Study II

3.1 Proposed architecture

To achieve our goal that the new architecture should have no API changes whatsoever, such that previous Hadoop applications can run seamlessly and enable full backward compatibility, we proposed the internal re-architecture of Hadoop that had an in-memory cache with a cache-aware scheduler. Caching needed to be done at exactly the right place of the MapReduce workflow. By adopting persistent-process-worker design, we ensured data-reuse of cache was blazingly fast.

As mentioned previously (Section 2.3), a lot of the proposed architecture was based on hindsight. The architectures were re-designed in such a way to incorporate a major chunk of the improvements/benefits as seen in section 2.3.1. Our initial architecture was based on Hadoop 1.x and is represented in Figure [3.2].

3.1.1 In-memory cache

The first necessity of caching, was to understand what exactly needed to be cached. As we saw in our background study, Hadoop deals with and shuffles a number of different sets of \(<K, V>\) pairs. If we cache the output \(<K, V>\) pairs after map operation, then we lose interactivity – i.e, cache based runs based on non-deterministic map functions or user-input map arguments cannot function properly. Caching post reduce is not useful, except for certain subset of algorithms which are iterative that depend on convergence the algorithm to finish. Caching of the input data is useful, especially since we have previous literature stating that 60% of production systems ingest the same data[2] over their Hadoop Jobs. But then again, that would store unnecessary non-stripped input data in-memory and is inefficient.
Chapter 3. Proposed Architecture & Study II

Figure 3.1: High level overview of proposed architecture

Figure 3.2: Detailed proposed architecture for Hadoop
We understood that parsed input data, that serves as input to map function of MapReduce, is the perfect set of $<K, V>$ pairs to be cached. They maintain interactivity in repeat Hadoop Jobs, while also stripping the input data to a compact form.

After studying previous research efforts, we understood that network-based access to cached $<K, V>$ pairs, or IPC access of cached $<K, V>$ pairs are slow and should be avoided. Individual access of $<K, V>$ pairs, and fine-grained updates/access is also slow. So we needed bulk cache objects of the parsed $<K, V>$ pairs, with them being cached in the worker-processes itself for very fast cache-access.

In Figure 3.1, we see that we use in-memory cache within the Map processes itself. The worker processes have been made persistent for easier cache-access. The cache is lazily backed up to the distributed file-system. The cache-aware scheduler manages the tasks assigned and is aware of the placement of the bulk cache-objects within the persistent Mapper process. We explain all of the above in the following sections.

The cache in-memory data-structure by itself is up to the implementation. In our implementation, as we'll see in the following chapter, we use a custom HashMap linked to a list of $<K, V>$ pairs. The HashMap key is the unique-identifier of the bulk cache-object (Section 3.1.4).

For Hadoop specifically, we found out through experimentation that caching at Mapper class was more performant than caching at RecordReader.

### 3.1.2 Persistency

The double-edged boxes in Figure 3.1 represent non-killable and non-terminating worker processes. By making the worker processes persistent, in-memory cache created in one Job can be easily accessed in a later succeeding Jobs – allowing easier cache sharing. Using this persistency model, we also ensured that the cache-access was local to the process, reading directly from memory and therefore very fast.

For Hadoop, the idea was to have the first Job spawn multiple Child processes, which were made persistent. These Child JVM(s), which execute individual Map Tasks[28], are not killed by the TaskTracker[29] of Hadoop. The code of Hadoop should be modified to ensure that later Jobs will re-use the already spawned Child task JVM(s).

This persistency enables sharing of any local data-structure(s) across Jobs. The first Job which spawns the initial Child processes stores key-value pairs that it parses from input files to an in-memory data-structure.
3.1.3 Cache-aware scheduler

With persistency enabled, it was now the scheduler’s responsibility to ensure maximum cache hit for later Job runs. The cache-aware scheduler, as represented in Figure 3.1, is aware of the location of bulk cache-objects stored in-memory. In succeeding tasks, when the scheduler encounters an input split that had been previously cached – it will try and schedule the task to the persistent worker-process that contains the cache. The internal scheduling algorithm, and methodology is implementation specific – and has been discussed to certain detail in the next Chapter.

For Hadoop specifically, with intelligent scheduling and InputSplit[16] - TaskTracker assignment logic, we could enforce the same InputSplit of a future Job to be assigned to the same persistent Child process which had previously parsed the split – thereby reading the <K, V> pairs directly from the in-memory data-structure.

The in-memory cache is only for Mapper related persistent Child process. With this new architecture, we’re looking at leveraging the parsing benefit, the in-memory benefit and reuse (of JVM) benefit.

Now, if a future Job-task has a cache miss, i.e. a particular Child task is assigned an InputSplit which it had not previously parsed – then we have a less than ideal scenario. We could workaround by using IPC (Inter-Process Communication)[30] among the persistent Child JVM(s) to share and exchange the <K, V> pairs; or even possibly allow lazy-write back of the in-memory objects to HDFS for the cache-miss Child process to read-back later. (Section 4.5.2)

3.1.4 Unique identifier

The bulk cache-objects need a mechanism for their identification and representation within the worker processes and the scheduler. The combination of input split parameters, format of input data, and input filename itself is assumed to be unique. Therefore, we compute an unique-identifier which uses the above parameters to uniquely identify a cache-object. The unique-identifier, depending on the implementation, is used to identify and retrieve the key-value pairs within the in-memory cache.

In Hadoop, each of the Child JVM’s would check if the uniquely named identifier for a particular combination of InputSplit[16] and InputFormat[17] exists in the in-memory cache. If it does exist, it will try and read the key-value pairs from them.

If the unique-identifier is not present as a key within the in-memory cache, then the new architecture will parse the InputSplit using the InputFormat’s associated Recor-
Chapter 3. Proposed Architecture & Study II

*dReader*. It will then store the \(<K, V>\) pairs in the in-memory data structure, using the unique-identifier as the key.

**Naming scheme example**

*InputFormat*: `HTTPInputFormat`

HDFS Input File Path: `/input/xaa`

*InputSplit* start byte: 0

*InputSplit* length: 98792342

```java
String unique-identifier = DigestUtils.md5Hex(HTTPInputFormat.getCanonicalName() + '/input/xaa' + '0' + '98792342');
```

### 3.1.5 Optional serializer

The list of \(<K, V>\) pairs for a particular input split could be serialized using an optional serializer as shown in Figure 3.2. With that, all the \(<K, V>\) pairs would now be stored in-memory as one big *Object*. The advantage of this feature is that access to all the \(<K, V>\) pairs would be possible with one read call. This is also a more compact form of representation resulting in lower memory consumption. The number of Java *Objects* within Hadoop would also be lowered, which is generally good for garbage collection. This, however, would mean further computation to deserialize the data. This tradeoff has been discussed to some extent in the evaluations (Section 5.2.1).

In Hadoop, a normal key-value data-structure would generally store a list of \(<K, V>\) pairs against a unique-identifier in the *HashMap*. This can be serialized using libraries such a Kryo[31]. We go a bit more into the implementation details in the next Chapter.

### 3.1.6 HDFS writeback of Cache

The cache-structure would be lazily written back to HDFS for backup, and to make space for new unique-identifier entries. The entries to be flushed to HDFS would be determined by the LRU algorithm[32]. In case of a *cache miss*, we first check with HDFS to see if we could possibly retrieve from the backup. (Figure 3.2)

But before we went ahead to actually implement the architecture, we wanted to get an approximate idea of the time benefits we would achieve using it. Therefore a set
of experiments were run, to evaluate the efficacy of storing Key-Value pairs in memory using Ramdisk[33] and SequenceFileFormat[34, 35].

3.2 Study II – Simulating in-memory key-value store using Ramdisk and SequenceFileFormat

In the previous study (Section 2.4), we investigated on how buffer cache improves the execution time of cached Hadoop Job ingesting the same input data.

Based on those results, we proposed a new architecture that believed that using an in-memory key-value store of Mapper[36] output would be beneficial for similar repetitive or pipelined [37] Hadoop Jobs. Retaining a persistent distributed in-memory $<K, V>$ store would effectively import some of the advantages seen in Spark[38], while maintaining the historic Hadoop API.

This backward-compatibility would enable industries to take advantage of their now new higher-RAM machine clusters to achieve speedup without having to rewrite their Hadoop Jobs, or re-deploying/re-architecting their data-centers.

Quick experiments using Memcached[39] intended to vaguely mimic our intended implementation did not produce positive results; but this was believed to be because of other reasons (over-the-network $<K, V>$ pairs access, pure distributed-memory store, etc.) rather than our re-architecture. (More on this in Appendix B.2).

Before going ahead with the implementation, we wanted to get an idea if the new architecture would be beneficial. Therefore we ran some experiments to measure the Hadoop Job runs using SequenceFileFormat as input. SequenceFileFormat, stores parsed $<K, V>$ pairs directly onto the file. By storing them on Ramdisk, and reading from it, we were vaguely mimicking our new architecture. This only took advantage of the in-memory benefit. We did a direct comparison with runs of SequenceFileFormat input read from hard-disk. We contended that if we are able to see considerable performance improvement with the ramdisk-based read vs the disk-based read, in conditions which were pessimistic (compute-heavy Job) in a harsh environment (only in-memory benefit); then our new architecture would perform incredibly better.

3.2.1 Experiments

To start with a baseline, we compared the read-rates of Ramdisk vs Hard-drive for various file-sizes. This gave a approximate idea of how much time is taken to read data of the
default split size (64MB). With this data, again, we could vaguely estimate the time-gain expected in a large-input Job. This gain does not take into account the parsing benefit. (Section 2.3.1)

**SequenceFileFormat**

*SequenceFileFormat* is an inbuilt Hadoop format which stores parsed *<K, V>* pairs. Raw input files, can be read and parsed by an *InputFormat*'s (e.g. *HTTPInputFormat*) *RecordReader* to produce *<K, V>* pairs. These *<K, V>* pairs can now be stored in the *SequenceFileOutputFormat* and can be read-in again by Hadoop *SequenceFileInputFormat*, without having to be parsed again.

There is some amount of computation involved with serialization and de-serialization of *SequenceFileFormat*s however. Doing away with parsing in the repeat *Jobs/tasks* is what we call the parsing benefit. This is the same as the parsing benefit discussed in Section 2.3.1.

Additionally, the *<K, V>* pairs tend to be of much lower size than each input line resulting in a much lower file input size for repeat *Jobs*. It might be worth noting that *SequenceFileOutputFormat*, which is used to write back to disk/Ramdisk, is compatible with the *SequenceFileInputFormat* used to read-back the same file.

In the Basic *SequenceFile* Experiment (Section 3.2.1.2), we try to do a comparison of *SequenceFile* read from Hard Disk over *SequenceFile* read from Ramdisk, the latter trying to simulate the new architecture. The hope would be to see the benefit of using Ramdisk, in-spite of Hadoop being a high-latency system.

Ramdisk access of *SequenceFiles*, can never be fast as a direct in-memory data-structure read of *<K, V>* pairs. This is because deserialization is required for *SequenceFile*. In addition to that, Ramdisk has the further overhead of system calls and filesystem read calls.

### 3.2.1.1 Read-rate experiment

In this experiment, files of varying sizes were read from both Ramdisk and Hard-drive. The averaged read-throughput and read-time values were collected to give an approximate idea of pure first-time reads of files from either in-memory/disk-based data source.

The linux standard *dd*[40] tool was used, with the buffer cache being erased after every run. The values were averaged over 5 runs. A server used for the experiment had 5400rpm Hard-drive and DDR3 1600Mhz SDRAM. The results in Table [3.1], [3.2] tabulates the speed (MB/s) or time-taken (seconds) for reads from either Hard-drive or Ramdisk, for varying file sizes. Generic observations are these:
Table 3.1: Speed (MB/s) for read of various file-sizes

<table>
<thead>
<tr>
<th>Input size</th>
<th>410KB</th>
<th>4.1MB</th>
<th>41MB</th>
<th>410MB</th>
<th>4.1GB</th>
<th>41GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramdisk</td>
<td>2048</td>
<td>1464.32</td>
<td>1812.48</td>
<td>1607.68</td>
<td>1873.92</td>
<td>2324.48</td>
</tr>
<tr>
<td>Hard-drive</td>
<td>34.33</td>
<td>341.67</td>
<td>845</td>
<td>150</td>
<td>155</td>
<td>140</td>
</tr>
</tbody>
</table>

Table 3.2: Time (sec) taken for read of various file-sizes

<table>
<thead>
<tr>
<th>Input size</th>
<th>410KB</th>
<th>4.1MB</th>
<th>41MB</th>
<th>410MB</th>
<th>4.1GB</th>
<th>41GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramdisk</td>
<td>0.00024</td>
<td>0.0029</td>
<td>0.024</td>
<td>0.25</td>
<td>2.26</td>
<td>18.59</td>
</tr>
<tr>
<td>Hard-drive</td>
<td>0.011</td>
<td>0.017</td>
<td>0.049</td>
<td>2.73</td>
<td>26.42</td>
<td>274.65</td>
</tr>
</tbody>
</table>

i) Time-reads for files of sizes ~ 64MB (default split size) from either Ramdisk or Hard-drive (HD), is quite low and similar in order of magnitude (~ .05sec or lower). In a high-latency system like Hadoop where each Map task is at least about 1 second in duration, the disk-read time is not significant.

ii) Time/speed values for file-sizes ≤ 64MB are quite inconsistent. This is expected since the disk access time is comparable in magnitude with read-time of small files. The time taken by disk-head to reach disk-sector gets amortized in larger reads.

iii) The read-throughput gets more consistent with larger file sizes. Hard-drive speeds settles at around 150MB/second while Ramdisk read speeds hovers around 1.6-2.0GB/second.

iv) To achieve any sort of significant speedup in an in-memory key-value store for Hadoop, the file split-size must be of the higher order and the number of Map tasks must be reduced. Figure 2.3

With these observations in mind, we now move on to our first real experiment where we could possibly calculate the expected speedup beforehand.

### 3.2.1.2 Basic SequenceFile experiment

In this basic set of experiments, a couple of custom Hadoop Jobs were run against varying size of input files as either raw-input, SequenceFile on Hard-drive or SequenceFile on Ramdisk.

The custom Hadoop Jobs, ModGrep and ModLogAnalyze, were slightly contrasting in nature – in the fact that ModLogAnalyze’s RecordReader parsing of raw input file was regular expression[21] based and hence computationally heavy.
Table 3.3: ModGrep – Job completion time (JCT)

<table>
<thead>
<tr>
<th>Input size</th>
<th>Raw input on HD</th>
<th>SequenceFile on HD</th>
<th>SequenceFile on Ramdisk</th>
</tr>
</thead>
<tbody>
<tr>
<td>10GB</td>
<td>265.5</td>
<td>16.22</td>
<td>17.66</td>
</tr>
<tr>
<td>5GB</td>
<td>133.98</td>
<td>15.7</td>
<td>18.12</td>
</tr>
<tr>
<td>2GB</td>
<td>66.86</td>
<td>15.4</td>
<td>16.7</td>
</tr>
<tr>
<td>1GB</td>
<td>43.86</td>
<td>15.44</td>
<td>15.88</td>
</tr>
</tbody>
</table>

ModGrep was essentially WordCount[41] for a given input search-term and therefore the output <K, V> pairs (or SequenceFile) were usually small (28KB for 1GB of raw input file). ModLogAnalyze took HTTP log files as input and summed the total bandwidth transferred for a particular user (of the format http://www.eecg.toronto.edu/~username) of a specific month of the year. This was more computationally complex and produced \(\sim\)135MB SequenceFile for 1GB of raw input file. ModGrep and ModLogAnalyze have been referred to in Sections 2.4.2.1 and 4.6.2.

The Hadoop temporary directory (hadoop.tmp.dir[42]), was changed between the Ramdisk or normal Hard-drive locations for different runs of the experiment. Since the Hadoop Distributed File System (HDFS)[13] used the hadoop.tmp.dir, the SequenceFiles were being read from Ramdisk (ramFS) or Hard-drive. For the first set of experiment, Raw input files were only read from the Hard-drive.

With 10x1GB, 5x1GB, 2x1GB and 1GB raw input(s), the custom Hadoop Jobs were run, timed and analyzed (using JobHistory[43] API). The values were averaged over 5 runs and graphed. It should be noted that for the SequenceFile-inputs the sizes were correspondingly smaller. For ModGrep that was (10x28KB, 5x28KB, 2x28KB, 28KB), while for ModLogAnalyze it was (10x135MB, 5x135MB, 2x135MB, 135MB). The SequenceFile-inputs were generated and stored in the HDFS beforehand.

We’re storing only the matched values in ModGrep. Words, that do not match the input search term are discarded. That is the reason why the SequenceFiles are much smaller. We’ll see much later (Section 4.6.1) a similar Hadoop Job that does not discard the unmatched values – allowing it to search for totally different input search terms in following runs and making it interactive.

Before the experiment runs, we were hoping that there would be some immediate small improvements between the latter two sets (SequenceFile Hard-drive vs SequenceFile-Ramdisk) at least for ModLogAnalyze (given the larger file-input size). The tabulated values and plotted graph are seen in Tables [3.3],[3.4] and Figures [3.3],[3.4]. Observations from the graphs are as follows:
### Table 3.4: *ModLogAnalyze* JCT

<table>
<thead>
<tr>
<th>Input size</th>
<th>Raw input on HD</th>
<th>SequenceFile on HD</th>
<th>SequenceFile on Ramdisk</th>
</tr>
</thead>
<tbody>
<tr>
<td>10GB</td>
<td>443.54</td>
<td>66</td>
<td>58.22</td>
</tr>
<tr>
<td>5GB</td>
<td>231.5</td>
<td>39.7</td>
<td>37.8</td>
</tr>
<tr>
<td>2GB</td>
<td>97.1</td>
<td>22.84</td>
<td>22.26</td>
</tr>
<tr>
<td>1GB</td>
<td>56.66</td>
<td>21.78</td>
<td>21.5</td>
</tr>
</tbody>
</table>

**Figure 3.3: Graph of *ModGrep* JCT**

**Figure 3.4: Graph of *ModLogAnalyze* JCT**
i) For all the experiments run (both ModGrep and ModLogAnalyze) there is significant difference in the execution time of Raw input vs the SequenceFile inputs. This is expected. This is a combination of the parsing benefit and that the SequenceFile inputs are significants smaller and hence run lesser number of Map tasks.

ii) In ModGrep the input size for the SequenceFile experiments are \( \leq 280K \), therefore it runs 1 Map task. Since the read-time for such an input size for both Hard-drive and Ramdisk is insignificant, the time-values we see for either of them are almost equal.

iii) In ModLogAnalyze, the input sizes in SequenceFile experiments are 1350MB, 675MB, 270MB, 135MB. The difference in execution times in Hard-drive vs Ramdisk are: 8sec, 1sec, 0.6sec and 0.2sec correspondingly. This falls slightly short of the expected values, as per the read-rate experiments, which approx. are: 8sec, 2.5sec, 1sec and .7sec.

### 3.2.2 Modified SequenceFileInputFormat experiments

Even though the basic SequenceFile experiments succeeded, we wanted to note how it would scale to an input of 1TB – if we would be able to see a similar performance improvement on the Ramdisk based run. A number of experiments were conducted (which have been moved to Appendix B.1.4 for the sake of brevity) – none of which met the performance that one one would expect from a Ramdisk based storage. In our final experiment in the Appendix B.1.4, we are able to conclude that SequenceFileInputFormat is not optimized for Ramdisk based read. Therefore, SequenceFileInputFormat will need to be modified, optimized for bulk-reads from in-memory filesystems. This section explains those modifications and the results of the re-run with 1TB of input data.

Two implementations were done of Modified SequenceFileInputFormat, both based on the SequenceFileInputFormat, with Implementation 2 extending Implementation 1 but drastically fine-tuning Hadoop parameters and stripping large quantities of the source-code.

In Implementation 1, three crucial points of the source code were identified where InputDataStream bulk-read was taking place using the HDFS FileSystem Java interface. This was modified to read larger-amounts of data. This also hinged on the io.file.buffer.size[44] parameter – which was increased to 4MB (from the default 4KB).

In Implementation 2, a large chunk of the source code – which were deemed unnecessary for this experiment – were stripped off. As a result, a lot of the extraneous disk-read calls were done away with. In addition to this, aggressive fine-tuning of Hadoop
Chapter 3. Proposed Architecture & Study II

<table>
<thead>
<tr>
<th>2GB Split (1350MB input files) and No Reduce Job Runtime (in seconds)</th>
<th>Implementation 1</th>
<th>Implementation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard-drive</td>
<td>2393</td>
<td>1454</td>
</tr>
<tr>
<td>Ramdisk</td>
<td>2253</td>
<td>1228</td>
</tr>
<tr>
<td>Improvement (in seconds)</td>
<td>140</td>
<td>226</td>
</tr>
<tr>
<td>Percentage of expected improvement (expected = 600 seconds)</td>
<td>23%</td>
<td>38%</td>
</tr>
</tbody>
</table>

Table 3.5: Modified SequenceFileInputFormat with 135GB input, ModLogAnalyze

configuration parameters[44, 45] was done. Briefly –

i) mapreduce.ifile.readahead – We turned off IFile readahead feature of Hadoop, which allowed Hard-drive to gain significant advantage in the experimental runs.

ii) io.bytes.per.checksum – In addition to removing the checksum feature in the source code, we proactively set this parameter to 16384 from a default value of 512.

iii) fs.checkpoint.period – The file-system checkpoint period was increased to once every 10 hours.

iv) local.cache.size – The local cache size for Hadoop was reduced to 0.

All the averaged-run results can be seen in Table 3.5. We repeated the last set of experiments from Section B.1.4, which is the 2GB split with 102 1350MB input files experiment – running ModLogAnalyze Job without the Reduce component. The expected performance improvement that we had calculated previously was 600 seconds for Hard-disk vs Ramdisk.

We observe in Implementation 1, the Ramdisk runtime is 2253 seconds, a gain of 140 seconds. The Hard-disk runtime is still around 2400 seconds (as we saw in Section B.1.4). Compared to the calculated 600 second performance gain that was expected, we achieved a 23% improvement.

In Implementation 2, the Hard-disk runtime now drops to ∼1450 seconds and the Ramdisk runtime is 226 seconds lower. So that means we’ve achieved ∼38% (226/600) of the expected improvement with this implementation.

Though we couldn’t modify the SequenceFileInputFormat completely to show the full 600 second benefit, we believe the improvements we see here is validation enough for the proposed new architecture.

With a very pessimistic baseline, using only the in-memory benefit, we were able to show a performance improvement of 15.5%. This again shows that with the parsing benefit, in-memory benefit, reuse benefit – all included – our new architecture would be able to perform much better.
3.3 Conclusion

In this chapter, we proposed a new re-architecture for Hadoop that took advantage of an in-memory cache and cache-aware scheduler. This new architecture hinged on the benefits that we observed from our background study (Section 2.3). The new architecture was enabled by a persistent-process-worker based design and had a vast featureset including a unique cache-identifying mechanism, an optional serialized-cache, and a HDFS-backed cache-propagation system. Maintaining backward compatibility with existing Hadoop applications is of supreme importance and is maintained in this architecture.

We conducted a set of experiments that vaguely mimicked the new architecture – but in a very pessimistic fashion. The experiment, taking advantage of only the in-memory benefit – tried to validate the architecture. We argued that if with this new architecture, we’re able to catch up with the same degree of improvement as 135GB of read from disk vs Ramfs – then our Hadoop Jobs would be as performant (if not more) in our proposed architecture.

From the last set of experiments with Modified SequenceFileInputFormat, we could see that the proposed architecture would be beneficial. Even in very unfavorable conditions – computationally heavy Job, shuffle and sort heavy operations – with only in-memory benefit we were able to observe improvements of upto 15.5% of JCT. We now have definitive proof to go ahead and actually implement the architecture.
Chapter 4

Implementation

Our prototype implementation followed the proposed architecture from figure [3.2]. Hadoop codebase turned out to be quite complex and coding the initial set of features took considerable amount of time.

One of the major considerations we had throughout the implementation was to ensure that none of the Hadoop API’s changed. So technically, what we were aiming for was for existing deployments of Hadoop to copy our new JAR (Java Archive) Hadoop-core files in-place, restart Hadoop, and notice a N-x improvement in Hadoop Job runtime. Apache Hive, Mahout, Giraph, and all other technologies using Hadoop as its base would directly propagate and imbibe the performance improvements. Retaining Hadoop’s strong security quotient was not prioritized in this prototype implementation.

Our implementation was on top of Hadoop version 1.2.1, which was the latest available 1.x version at that time. The most important feature that was added was to make the Child process persistent across Jobs. We then modified the Mapper to store and share \(<K, V>\) pairs which were initially parsed by the RecordReader. Later, we moved to a type-free implementation of the Mapper. This ensured that our code worked well with all possible application-classes of \(<K, V>\) pairs.

Then we started scaling up to multiple Child processes, and to multiple machines (nodes). We modified the architecture slightly to run multiple TaskTrackers on a single machine to ease the coding effort of our prototype implementation. This has been discussed to certain amount of detail in Section 4.3.

Since each of the InputSplit (Section 2.2.7) is identified using the unique identifier mechanism (Section 3.1.4), and because each InputSplit is being parsed and stored in a particular Mapper, we had to ensure that each of the follow-up Map tasks for the same InputSplit-InputFormat combination is assigned to the same particular Mapper. Therefore, we modified JobTracker’s scheduler.
We added other features to our implementation, including an HDFS-backed LRU cache mechanism for the in-memory store, a serialized version of the in-memory store and other features that we discuss in the later subsections of this chapter.

The final sections of this chapter goes into internal details of the sample Hadoop Jobs that were used in previous experiments and upcoming evaluation. The last section speaks about how a crucial stress-test helped us understand, narrow down and figure out a particular bottleneck of our implementation – and that everything boils down to how well the user-level application code is written.

4.1 Persistent Child

The first feature to be implemented was also the highest in terms of code-complexity.

To make certain that the Child process JVM does not exit/die at the end of all of its assigned Map tasks, we had to disable all the possible kill workflows that existed. For e.g., after JobTracker notifies the TaskTracker that a particular Job has completed, it instructs the Child JVM to exit using the RPC-getTask() method. Similarly, there are other areas in the code where it actually does a process kill using Linux signals. All of these had to be disabled, worked around and modified.

Some of the internal data-structures and HashMap’s within the TaskTracker that tracked relationships between Job-Identifier\(\leftrightarrow\)JVM-Identifier, JVM-Identifier\(\leftrightarrow\)Process-ID, JVMRunner\(\leftrightarrow\)JVM-Identifier, etc. had to be studied and the workflows involving them had to be mapped out. They were altered to establish a smooth flow when a different Job-Identifier (JobID) tried to assign a new task on a pre-existing persistent JVM (that was spawned by a previous Job).

By modifying a key function reapJVM() within the context of a class that managed running JVM’s (JVMManager) – we tweaked the code path that determined whether a new JVM should be spawned or if it is possible to reuse an already spawned (but idle) JVM. TaskTracker’s getTask() was another crucial function that decided if an already persistent JVM (belonging to an expired Job) could be re-assigned a new Job’s task.

As mentioned above, certain security issues were not prioritized in this implementation. authorizeJVM() and validateJVM() which verified the sanctity of the relationship HashMap’s were essentially commented out. JobToken, a file-based security token mechanism used at the initial stages of Child process spawn, was also worked around for a consistent JobToken-value. As a result, concurrent Jobs were not supported in this prototype.
Chapter 4. Implementation

4.2 In-memory cache and Generic version

The core Mapper class now contains the in-memory cache storing the $<K, V>$ pairs. The in-memory data structure is a custom HashMap, where the value is a list of parsed $<K, V>$ pairs, while the key is the unique-identifier generated by the naming scheme (3.1.4) for the particular InputSplit-InputFormat combination. (Figure 4.1)

If the particular unique-identifier is not found in the list of Keys, then the Mapper uses the actual RecordReader associated with the configured InputFormat to parse and obtain $<K, V>$ pairs. It stores the pairs into the in-memory structure while passing it on to the main map() function. In case the unique-identifier is found, a cache hit, the list of $<K, V>$ pairs are read from the in-memory structure, looped-over and fed into the map() operation.

We had a generic type-free implementation of the in-memory store. i.e. our implementation was such that it is InputFormat and RecordReader-agnostic. It would work perfectly well with any custom InputFormat, RecordReader or Mapper extension that an application developer might wish to have. This was important to preserve the API integrity of the Hadoop codebase.

Our current technique just casts $<K, V>$ members to Java Objects, while an advanced
Java *Generics* and *Reflection* based realization might have made this a wee bit more performant.

## 4.3 Multiple *TaskTracker*

We deviated a bit from standard Hadoop architecture for our prototype implementation – instead of one *TaskTracker* per node, we decided on a multiple *TaskTracker* per node architecture. We also limited the number of *slots* or number of *Map/Reduce Child* processes that a *TaskTracker* could spawn to just 1.

We did this purely for easing the coding effort required, since we were eager to validate our architecture – and then later move to a better single-tasktracker per slave implementation once we had positive results.

With this change, our scheduling requirements was much easier. We just had to ensure that the scheduling was done perfectly at the *JobTracker* level, by taking into account previously cached/pre-parsed *InputSplits*. If we had retained the *TaskTracker*↔multiple *Child* architecture, we would have had to modify the scheduling (task-assignment code) at that level as well.

To run multiple *TaskTrackers* on a slave, we ensured that they operated on different working directories in the same machine. The *TaskTrackers* listened on different randomized port numbers. All of the start/stop Hadoop scripts were changed to dynamically start $N$-number of *TaskTrackers* per slave. Each *TaskTracker* now allowed a maximum of 1 *Map Child* process and 1 *Reduce Child* process. The JVM Heap size was increased to support this new requirement.

The impact of going this easier route, was that it affected how well we could scale. With some initial experimentation that we performed, we couldn’t scale as much as we wanted to. We were able to run about 4 *TaskTrackers* (and corresponding number of *Child* processes) concurrently. For our experiments and evaluation we took a conservative 2 *TaskTracker* per slave approach.

## 4.4 Scheduler changes

With the multiple *TaskTracker* architecture, scheduling *cached* tasks were much easier. Scheduling logic had to be modified only at the *JobTracker*↔*TaskTracker* level. We altered the default FIFO (First-in First-out) scheduler of Hadoop to take into account the in-memory cached objects that had been created by previously completed tasks.

In essence, the scheduler approach was this:
JobTracker identifies an unassigned task.

- It computes the unique-identifier using the naming-scheme (3.1.4)
- JobTracker checks if this task was previously parsed at some other persistent Task-Tracker-Child.
- When that particular TaskTracker sends a heartbeat to the JobTracker requesting for a new task, we assign this task.

To facilitate this, we had to introduce some new data-structures and mappings in the context of JobTracker:

- identifierCache and identifierReverseCache – At the beginning of a Job, the JobTracker pre-computed and stored the unique-identifier of each of the tasks.
- identifierToTaskTrackerMap and taskTrackerToIdentifierMap – Mapped which TaskTracker had cached an InputSplit-InputFormat combination of a previously run task. It was identified by the unique-identifier (3.1.4).

The new findNewMapTaskCached() function in the modified FIFO scheduler executed before all other possible task-assignment-functions. It greedily assigned and prioritized the tasks to those TaskTrackers which had the \(<K, V>\) pairs cached in the Mappers. findNewMapTaskCached() trumped all other locality-based scheduling logic in place when cache-hit was assured.

We also implemented delay-scheduling[46], i.e. we proactively delay assigning a particular free TaskTracker-Child an unassigned remaining task.

This was done if we had prior knowledge that another TaskTracker-Child, which is not currently free – but soon might be –, already contains the cached \(<K, V>\) pairs related to the task. Therefore the latter TaskTracker-Child will be able to execute the task faster – lowering the overall Job completion time.

### 4.5 Other implementation notes

#### 4.5.1 Garbage Collector impact

We noted that after the initial parse of a big input file and creation of a very large number of in-memory cache objects, a relatively long JVM Garbage Collection (GC) cycle got triggered. A sample 1GB input file could create 136million objects (4.6.1.1).
GC, normally, is triggered automatically by the JVM. Because of this unpredictability it could interfere with our experimentation, which is why we forcibly ran GC before each of our cached evaluation experiments.

### 4.5.2 HDFSBacked-LRUCache

Our implementation stores the in-memory objects identified by the unique-identifier (Section 3.1.4) based on the LRU[32] algorithm. The list of \( <K, V> \) pairs which remain unused over large period of time is written-back to HDFS. Thus the name *HDFSBackedLRUCache*. (Figure 4.1)

In case of a *cache miss* in the *Mapper*, we first check with HDFS to see if we could possibly retrieve it. This feature was not tested in our evaluations since delay-scheduling was prioritized.

### 4.5.3 TaskTracker retain of files

In our prototype implementation, because we ran multiple *TaskTrackers* per slave, left *Job* files and JAR-files in place at the end of *Job complete* instead of deleting them. This was done to resolve Java classpath issues on repeat-cached runs of experimental Hadoop *Jobs*.

### 4.5.4 Serialized versions

As noted in Section 3.1.4, a unique-identifier was mapped to a list of \( <K, V> \) pairs in our custom *HashMap* in-memory store. In a parallel implementation, we tried to serialize the list of \( <K, V> \) pairs and store them in the *HashMap*.

Using a very fast serializer/deserializer such as Kryo[31], we noted that the total number of Java *Objects* in the JVM drastically decreased. This resulted in near-zero Garbage Collection times. Memory consumption would also be lower in a serialized representation of the in-memory cache.

But, this also meant that deserialization needed to take place to retrieve back \( <K, V> \) pairs on a cached-run – which made it slower by 1-1.5x. We notice the tradeoffs of using this serialized version in our evaluations (Section 5.2.1).

### 4.5.5 Optional Cleanup, Setup

*Job Cleanup* and *Job Setup* tasks are custom Hadoop tasks which help application developers to perform trivial housekeeping/setup tasks before and after the *Job* is run. They
were not useful for our evaluations and therefore were disabled using patches[47].

4.5.6 From high-latency to low-latency

Hadoop was initially engineered keeping in mind that it was a high-latency system. This meant that Jobs would have completion times on the order of minutes. Therefore, allowing sleep() calls amounting to second(s) or hundreds of milliseconds within the Hadoop code was not detrimental.

We saw that in many code-instances, and this led to a general slowdown and hog when we switched to our implementation. We had map() run time in the order of seconds – and therefore having sleep() time as high as the previous values was just unacceptable.

As a result, we reduced sleep() time at various carefully chosen code-instances, to between 10 – 100/1000th of a second. By this, we were attempting a low-latency system for our new architecture.

4.6 Hadoop Jobs

In this section, we discuss about the two major Hadoop Jobs that we used for our evaluation.

4.6.1 WikipediaSearch

This Hadoop Job takes an XML dump of Wikipedia as input, usually spliced into a specific N-GB size limit, and searches for an input term provided by the user. The searches are case-insensitive.

For our new architecture, repeat WikipediaSearch should be fast for any search term. i.e. even for different search terms in repeated runs, the cache based run should be fast. That means that in-memory cache is independent of the search term and therefore is interactive.

The input file data is not stripped at the RecordReader and stored in the Mapper based on the search term. But rather, all the words/lines from the input-file are transformed into <K, V> pairs at the RecordReader and stored in-situ the in-memory data structure.

The general approach of this Job is to let the Hadoop native LineRecordReader (which extends the abstract RecordReader) parse the input file and provides <‘Line’, 0> pairs to the Mapper. The Mapper caches the <‘Line’, 0> pairs against the corresponding
Chapter 4. Implementation

Figure 4.2: Internal-tokenizer vs External-tokenizer and Flow of $<K, V>$ pairs in WikipediaSearch

*InputSplit*-*InputFormat* unique-identifier (3.1.4) while passing it as input to the *map()* function.

The *map()* function splits (tokenizes) the ‘Line’ into ‘Word’s, matches (in a case-insensitive fashion) to the input-term and writes $<’Matched-word’, 1>$ pairs to the Reducer in case of every match. The reducer *reduces* (in this case, sums up) all the matches. It writes back the final $<’Matched-word’, ’sum of matched words’>$ into HDFS-output.

There are two different ways to solve this MapReduce problem, both of which are explained in the upcoming section.

### 4.6.1.1 External tokenizer vs Internal tokenizer

External Tokenizer (or Line-granularity approach) tokenizes the ‘Line’ in the *Mapper* class within the *map()* function. The *Job* pretty much follows suit as described in the section above. The in-memory cache contains $<’Line’, 1>$ pairs. A sample 1GB Wikipedia file input creates about 11 million $<’Line’, 0>$ pair objects. Since repeated-runs of search requires redo of tokenization of ‘Line’s, this is considered slow. Internal Tokenizer (or Word-granularity approach) tokenizes the ‘Line’ at the *RecordReader* itself, and stores $<’Word’, 0>$ pairs in the *Mapper* in-memory cache.

The *map()* function has to only perform the case-insensitive match against each of the stored words. Since tokenization is eliminated for repeat *Jobs*, *map()* is much faster.

Consequently, this stores about 136 million objects for the sample 1GB input. This has a significant GC impact, since GC increases with increased number of objects. The amount of memory consumption is also higher. Since we assume that memory is not constrained in today’s hardware, we prefer this approach since it’s much faster in the
We’ll see in the next section 4.7, and in early evaluations that Line-tokenizer is slower in its cached run. (Table 4.2)

A figurative representation of how the <K, V> pairs interact with classes in both the implementation is shown in Figure [4.2].

4.6.2 Log analyzer

The log analyzer used in our evaluation is the same as the ModLogAnalyze Job introduced in Section 3.2.1.2. It is much more shuffle and reduce intensive than WikipediaSearch. Its initial parsing uses a custom RegexRecordReader (regular expression based RecordReader) which makes the parse and cache-build compute intensive.

The new implementation for our evaluations differs slightly from the Job discussed in section 3.2.1.2 in the fact that this Hadoop Job tries to mimic an interactive query. i.e. For a query that poses the question – “From the HTTP logs, which user-id consumed the maximum bandwidth of our servers for the month of July?” (Figure 4.3) – cached ModLogAnalyze is flexible enough to answer the question using the cached-data even when the user inputs a different month/user-id.

Therefore, the custom RecordReader for ModLogAnalyze ensures that the <K, V> pairs it passes on to the Mapper is all-inclusive, and that it does not strip data based on the month-matching. A generic non-interactive Log Analyzer would’ve matched the month “July” at the level of the RecordReader and discarded the remaining input data. So, the in-memory cache built would only contain <K, V> pairs for the month of “July”. A repeat query for the month of “August” would mean re-parsing of the input data. This is inefficient and not desirable in the new architecture. Keeping this in mind, our new Log Analyzer implementation did not discard input data at the RecordReader. The

“Which user-id consumed the maximum bandwidth of our servers for the month of Jun?”

Figure 4.3: Log analyzer Job

cached runs.
<K, V> pairs stored in-memory was for all months included – ensuring that it is truly interactive for repeat queries.

## 4.7 Stress test & Optimization

The Hadoop Job needs to be written by the developer keeping in mind the new in-memory cache. In this section, we’ll see how changing even the intermediate <K, V> format of a particular Hadoop Job greatly affected Job completion time. We’ll also see the crucial stress test that helped us find the aforementioned bottleneck.

One of our early implementations, consisted of only the non-generic in-memory cache and persistent Child. It did not possess advanced features like multiple TaskTracker support (Section 4.3), HDFSBacked-LRUCache (Section 4.5.2) etc. We ran a sample experiment with it against WikipediaSearch for 1GB input. i.e. search a particular input term in a 1 GB Wikipedia dump.

We also wrote a standalone single-process single-core Java application that mimicked the in-memory cached search functionality of WikipediaSearch. In the source code provided in Appendix A, the first phase reads all the words/lines from the 1GB input file and creates an in-memory data-structure which is exactly the same as the structure used within the Mapper class. In the second phase, which is almost equivalent to the Mapper’s map() function, the values of the in-memory data-structure are looped through and checked for a match with the input term – and a counter is incremented if the word is actually encountered.

We wrote two major versions of the standalone program: one that was equivalent logically to Word-granularity (Internal Tokenizer) version of WikipediaSearch. The second one was equivalent logically to the Line-granularity (External Tokenizer) version of WikipediaSearch. The latter (Line-granularity) version cached lines. The lines had to be always tokenized to words in the 2nd phase and therefore is slower. The 2nd phase, which performs the in-memory search, was timed.

When performing a direct comparison between the Hadoop Jobs and the standalone program, we noticed that the Hadoop Jobs were much slower. A word-granularity standalone program was taking 6.88 seconds to perform the in-memory search, while our WikipediaSearch with word-granularity took 50.49 seconds, with about 46.33 seconds being spent on map() operation. This was inefficient.

We then probed a bit into the Mapper class, reduced the creation of certain unneeded Objects in the cached run and lowered the runtime to 27.29 was – which was still much higher than 6.88 even if we take Hadoop orchestration time into consideration.
What followed was a series of iterative experiments and stress-test to narrow down to
the location of code where WikipediaSearch was lagging behind. A number of experiments
were performed, but here we only provide the gist of it.

The first experiment was a stress test that text-matched \(\sim 136\) million \textit{Objects}, since
that was the number of \textit{words} in 1GB Wikipedia dump. We did a quick standalone-
program comparison of \texttt{String.compareTo()} vs \texttt{String.equals()} for 136 million \textit{String Objects} – it showed that \texttt{compareTo()} was twice (6.59sec vs 3.84sec) as slow as \texttt{equals()}. We
got similar results comparing \texttt{String.compareToIgnoreCase()} vs \texttt{String.equalsIgnoreCase()}.

If we used Hadoop’s native \texttt{Text.compareTo()} instead of \texttt{Text.toString().equalsIgnoreCase()},
WikipediaSearch runtime was 9.60 seconds – however the comparison was case sensitive
and didn’t have as many matches as the latter.

This prompted us to do another test involving the creation of 136 million objects and
timing it. The results are in Table 4.1. As we can see, creation of \texttt{Text} is very heavy and
\texttt{Text.toString()} as an operation is pretty time consuming.

When we modified WikipediaSearch to use \texttt{<String, Long>} pairs instead of \texttt{<Text, LongWritable>} pairs which were native to Hadoop, we saw an immediate performance
improvement. WikipediaSearch now only took 9.93 seconds (Table 4.2), which was much
closer to the 6.88 seconds that a standalone word-granularity program took. The Hadoop
orchestration (\textit{Task} creation, heartbeat mechanism, \textit{Child} task-assignment, etc) is re-
sponsible for the other (9.93-6.88) \(\sim 3\) seconds. We also noted that GC (Garbage Collec-

<table>
<thead>
<tr>
<th>Test Description</th>
<th>Time taken (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Text.compareTo()} for 136 million Objects</td>
<td>6.59</td>
</tr>
<tr>
<td>\texttt{Text.equals()} for 136 million Objects</td>
<td>3.84</td>
</tr>
<tr>
<td>\texttt{Text.toString()}</td>
<td>28.98</td>
</tr>
<tr>
<td>Creation of 136 million \texttt{new Text(“a”)} Objects</td>
<td>16.75</td>
</tr>
<tr>
<td>Creation of 136 million \texttt{new String(“a”)} Objects</td>
<td>1.51</td>
</tr>
<tr>
<td>Creation of 136 million \texttt{new LongWritable(0)} Objects</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 4.1: Stress test with 136 million objects

<table>
<thead>
<tr>
<th>Test Description</th>
<th>Time taken (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial implementation, Word-granularity (1)</td>
<td>50.49</td>
</tr>
<tr>
<td>Standalone Program, Word-granularity</td>
<td>6.88</td>
</tr>
<tr>
<td>Standalone Program, Line-granularity</td>
<td>13.85</td>
</tr>
<tr>
<td>Mod of (1), forced GC, less \textit{Objects} (2)</td>
<td>27.29</td>
</tr>
<tr>
<td>Mod of (2), with \texttt{Text.compareTo()} (3)</td>
<td>9.593</td>
</tr>
<tr>
<td>Mod of (2), with \texttt{&lt;String, Long&gt;}</td>
<td>9.93</td>
</tr>
</tbody>
</table>

Table 4.2: Stress test, standalone program (A) and early WikipediaSearch
tion) had a significant impact (Section 4.5.1) in the in-memory cached runtime – so we forced a GC before the 2nd phase run of the standalone program.

What we learnt from this set of initial experiments is this – application code had a significant impact on how quickly the cached-runs executed. The application developer’s design and code choices, even as simple as the wrong data-structure, could adversely affect runtime. Luckily for WikipediaSearch, we were able to profile and map the timing at an early stage. Even the most minor of engineering decisions had major impact.

For the ModLogAnalyze code however, we did not perform any optimization, and plugged in the Hadoop Job directly on top of our new architecture. This probably served as real-life industry example – where people are more averse to code changes in a production system.

4.8 Conclusion

Our implementation based on Hadoop-1.2.1 closely followed the proposed architecture. Special care was taken to make this prototype backward compatible and not unwittingly introduce API changes. This was one of the most important technical consideration.

The implementation is as generic as possible, meaning that it would allow all possible \(<K, V>\) formats to be stored in the in-memory cache. The worker processes were made persistent – which was one of most defining features of this implementation. In addition to this, we implemented the scheduler algorithm within the JobTracker, scaled up to multiple cores using many TaskTracker instances within a single node and made a parallel implementation using Kryo[31] to serialize the cache. Other technical bottlenecks, now more obvious with the superfast cache were also removed.

In this chapter, we also introduced the two main Hadoop Jobs – WikipediaSearch and ModLogAnalyze that we use for our evaluations. Our implementation kept both the Jobs interactive.

We also performed a stress-test with the initial implementation and Hadoop Jobs. We understood that application code had a significant impact on cached-run execution time. An Hadoop Job code’s simplest design choices could adversely affect runtime.

Our prototype of the new architecture only briefly touched on the potential of this architecture. Future work, including a refined thread-based implementation, automatic scaling and zero-copying serialization mechanism for better storage of the cache will be very helpful in realizing all the benefits.
Chapter 5

Results & Evaluation

5.1 Methodology & configuration

We used three separate deployment configurations for our evaluations. WikipediaSearch (Section 4.6.1) was run more extensively on all possible configurations, while ModLogAnalyze (Section 4.6.2) was run mostly on the scaled configurations. We directly compared our execution time (JCT) with unmodified Hadoop-1.2.1 and Spark-1.0.0[1].

The two machines used for the deployment configurations were these:

- **Fat Node** – The larger of the machines available at our disposal had 512GB of RAM, with 64 Intel Xeon 2.20GHz cores running a CentOS 64bit version of Linux. At most two cores was used by the *Mapper Child* processes, even in our scaled experiments (Section 4.3).

- **Light Node** – In our multi-node scaled experiments, we used 10 of our *Light* nodes – which each had 32GB of RAM, and 32 cores of Intel Xeon 2.20Ghz processors. They all ran the same 64-bit CentOS Linux operating system. Here again, at most 2 number of cores was used for the *Mapper Child* process.

The deployment configurations:

i) The **single-core, single-node** configuration – We used a pseudo-distributed mode[48] of Hadoop on the *Fat* node, and restricted the *Mapper Child process* to only one core. The input size was set to 1GB.

For the Spark experiments using the same configuration, Spark ran in a standalone mode, using the same HDFS for input. We also restricted Spark’s usage of processor cores using its configuration parameter.
ii) The **two-core, single-node** configuration – Moving slightly away from the previous configuration, we increased the number of TaskTrackers in the Fat node to two, allowing the Mapper tasks to use at-most 2 cores. Spark’s configuration parameter was increased correspondingly. Another set of experiment was done using the same deployment configuration, but with 5 times the initial input (5GB).

iii) The **two-core, ten-node** configuration – We used 10 Light nodes as Hadoop slaves or as Spark workers, with the Fat node acting as the master. We allowed at-most 2 cores of each of the slaves to be used by the Mapper, while using a 10GB file as input from HDFS. The nodes were interconnected with an ethernet link at 1GBps.

We did not scale upward of 2 core per node, because we wanted keep the amount of data processed per core to be a realistic 512MB. If all the cores were to be used, the difference in the run-time would be very meagre, and therefore would not be captured. Additionally, we had a technical limitation of a maximum of 4 cores per node as explained in Section 4.3.

As explained in Section 4.7, the WikipediaSearch-code went through various stages of initial experiments, optimizations, profiling to remove bottlenecks since it was the first prototype-based experiment. The ModLogAnalyze-code however, was not changed at all – and was run unaltered without any optimization for the new architecture.

We did a one-to-one comparison of the Job completion time (JCT) required for the new proposed architecture, the unmodified vanilla Hadoop-1.2.1 and Spark-1.0.0. The comparison was done for both WikipediaSearch and for some of the scaled ModLogAnalyze.
experiments. Our implementation was based on Hadoop-1.2.1.

We used Spark’s Java API to write their equivalent *WikipediaSearch* and *ModLogAnalyze* code. Special care was taken to ensure that the code was logically equivalent to Hadoop’s *Job* code. We also measured peak memory-usage, and compared it with Spark for one of the smaller experiments. Spark also read the input files from HDFS. Each of the *Job* completion time (JCT) values were averaged over last 5 runs (or loop/iterations). The comparison was done and graphed against repeat (cached) runs.

For initial Hadoop-based evaluations, we experimented with the *Split* size configuration parameter. As seen in the following sections, split sizes play an important role in determining the cache-size per process and runtime. Spark split sizes were automatic, and usually tend to be 64MB per task.

### 5.2 Singe core, single node *WikipediaSearch*

The single-core, single-node *WikipediaSearch* experiment was the only evaluation which was heavily optimized through iterative profiling – therefore we were able to catchup with the standalone-program as explained in section 4.7. The *WikipediaSearch* was carried out at the level of Word-Granularity (Section 4.6.1.1).

**Split Size choice**

A normal unmodified Hadoop installation took 151.22 seconds to search a sample query term against the input 1GB of Wikipedia dump. This was when there was only 1 *Mapper Child* process which read the whole 1GB input split. Unmodified Hadoop has no concept of a *cached* run since every run read input files from the HDFS.

When we decreased the split size to the Hadoop default of 64MB, the total execution time increased to 225.84 seconds. There were a total of 16 *Map* tasks for this experiment. This amounts to approximately 4 second/task for unmodified Hadoop’s task scheduling and orchestration. Figure [5.3]. With our modified architecture, we performed the same cached search experiment in 13.23 seconds, which is almost a 12x improvement. Figure [5.4].

Split size has an effect on run-time, and larger splits meant lower number of *rounds* of scheduling. Larger splits, as we see in this and some of the following experiments, resulted in lower JCT. Therefore, we explicitly set bigger split sizes for all our following evaluations. (Figure 2.3).

The same experiment with Spark, with similar code-logic, took 10.95 seconds. It used Spark’s Java API to read the same file from HDFS in an networked environment. Our
Figure 5.3: 64MB vs 1GB split input WikipediaSearch JCT

Figure 5.4: Single-core, single-node 1GB input WikipediaSearch JCT
implementation loses out on task-scheduling, orchestration, Hadoop’s inherent heavy architecture and other factors resulting in the $\sim 2$ second difference. Figure [5.4]

**WikipediaSearch** with External-tokenizer

The speedup we see for Line-granularity *WikipediaSearch* is close to 2x, as seen in Figure 5.5. The overall speedup, for a best-case unmodified Hadoop vs best-case cached-Hadoop is 41.86 vs 13.23 (3.1x). Line-granularity approach, which is more compute intensive and non-ideal with respect to our cache-architecture, still gives a 1.9x speedup when compared to unmodified Hadoop – going to say that even the most non-optimized unaltered existing Hadoop *Jobs* can expect a 2x speedup with our new architecture.

In all the *WikipediaSearch* experiments, the number of matched values for the sample search term “india” was 3677. This meant a very efficient shuffle and a very quick reduce phase. The calculated *reduce()* time was around 40-60ms. Even when the sample search term was changed in the cached runs, we saw a similar 12x improvement – showing that our new architecture supported truly interactive *Jobs* and performed equally well.
5.2.1 Memory usage & Serialized version performance

We monitored memory usage in a separate run of the above experiment. We did this to do a head-to-head comparison of peak memory consumption. Though the in-memory cache memory usage is no longer a concern in today’s hardware, we did not want our implementation’s memory consumption to run overly astray.

A quick introspection showed that for an input 1GB file, at an average of 8 characters per word for \(~136\) million words, each word will have a \(<\text{String, Long}>\) pair and it’s corresponding reference (from the custom \textit{HashMap}) in the in-memory cache. This, when calculated, would require about 8.5GB of conservative RAM usage. In our experiment we observe that the amount of memory consumed is 14.3 GB of RAM, post GC, which is higher than the calculated value. This measured value includes the whole of the Child process, which, without the cache consumes less than 100M.

Similarly, we measured the peak memory consumption for Spark[1] worker to be 10.6 GB of RAM. Spark currently does a much better job of memory management.

Switching to a fast serializer/deserializer Kryo[31], for our implementation, the memory usage decreases drastically to 11.4 GB of RAM (Figure [5.6]). The forced GC that we perform before every evaluation measurement, was also reduced to less than a second.
This indicates that the number of *Objects* stored in the in-memory cache is now very low (ideally just one big serialized *Object*). With Kyro[31] however, there was the requirement of deserialization on cached runs, which did steal a number of CPU cycles. This resulted in the single-core single-node *WikipediaSearch* experiment taking about 19.42 seconds (Figure [5.7]), which is about 6.19 seconds slower. It is, however, still 7.8x faster than unmodified Hadoop.

So, even though our implementation was not targeted for optimal memory usage, we fared very well. Usage of the serializer/deserializer is optional and has its own tradeoffs. Memory consumption and the number of Java *Objects* is much lower with it – resulting in a very quick GC. However, the cached-run using Kryo is about 1.5x slower than our best performant cached-run. Kryo, again, was just plugged onto our implementation – more optimizations could be possible to improve its performance. We see one such suggestion in our Future Work (Section 7.2).
5.3 Two core, single node experiments

5.3.1 1GB Input, *WikipediaSearch*

In this experiment, for unmodified Hadoop, we increased the number of *Mapper Child* process that could be spawned to two. This allowed two concurrent *Map* tasks at the same time, allowing a total of two cores being used.

With a split size of 512MB, each of two *Mappers* processed exactly one split. It completed at an average time of 102.06 seconds. This is down from the ~151 seconds that it took in the previous experiment with unmodified Hadoop that we saw in the last section. Scaling is not linear for unmodified Hadoop, as we can see here.

In the persistent in-memory Hadoop architecture, for 512MB splits running with two *TaskTrackers* on the same node, we were able to complete the search within 8.79 seconds. This configuration allowed us to scale to 2 cores for the *Child Mapper* processes. 8.79 seconds, is down from the 13.23 seconds JCT in the 1-core run. Scaling, for in-memory persistent Hadoop is not linear, as of now. The speedup compared to unmodified Hadoop is again close to 11.6x.

With Spark, the total runtime was 6.07 seconds. They had automatic splits of 64MB.
each, and therefore created 16 tasks. Spark outperformed our implementation by about 2.72 seconds. We attribute this to the heartbeat mechanism and orchestration of Hadoop which is not as responsive as Spark’s event-driven Akka[49] architecture. This combined with the fact that every task in Spark is spawned as a new thread – makes it quite nimble. Scaling for Spark is not linear [6.07s two-core vs 10.95s single-core].

We scaled Spark with the two different methods for this experiment: 2 Workers using 1 core each or 1 Worker using 2 cores. The former spawns two separate JVM process while the latter uses 2 threads. We noticed that the latter method, the recommended scaling route according to Spark documentation, was faster by about 0.11 seconds at 6.07. We will rely on the latter method – Spark’s auto-scaled single-worker-process configuration – for all further upcoming evaluation sections.

5.3.2 5GB Input

5.3.2.1 WikipediaSearch
Scaling the same experiment further up to an input size of 5GB, the number of Map tasks was increased to 10 – with 5 tasks per core. The split size remained the same at 512MB.

Unmodified Hadoop now finishes in 528.53 seconds, while our new architecture completes in 47.84 seconds – providing a 11x speedup. We outperformed Spark, which took 67.34 seconds to complete. This is slower by 19.5 seconds – which makes us faster by 28.9%.

We almost scaled perfectly from our previous experiment (5 x 8.79sec ~44 seconds). Spark however, doesn’t perform well and scale. We were not able to ascertain the exact reason as to why the Spark Job completion times were high, but the logs indicate that for large amount of input data and splits, ShuffleMapTask is much slower for latter splits. ShuffleMapTask is Spark’s internal implementation for handling map tasks which partitioning and shuffling enabled.

5.3.2.2 ModLogAnalyze
We see a similar 11x speedup for ModLogAnalyze when run with 5GB input run over 2 execution cores. Spark, here, outperforms our implementation by about 10.82 seconds.

As mentioned before – ModLogAnalyze, unlike WikipediaSearch, was run ad-hoc over our new architecture. This was done just to get an understanding of how a previously compiled, non-optimized, existing Hadoop Job would perform on persistent in-memory Hadoop implementation without any modifications to the code whatsoever. We could
Figure 5.9: Two-core, Single-node 5GB input WikipediaSearch JCT

Figure 5.10: Two-core, Single-node 5GB input ModLogAnalyze JCT
have, probably, saved a couple of more seconds of execution time we had done a profile of the Job and optimized it further.

We believe that Spark is better at task scheduling, since it’s event-driven. Hadoop has 5 rounds of scheduling in this experiment (10 tasks over 2 cores, one after another), this might have introduced some lag given its heavy heartbeat mechanism. There is also much higher amount of shuffle and reduce in this experiment, which is not optimized at all and this could be improved upon as well. We reduce this performance gap in our ten-node experiments.

5.4 Ten node, two core experiments

In these experiments, we move to a multi-node deployment of Hadoop/Spark for evaluations. The input data size was increased to 10GB.

Even though the input data had increased, the amount of data processed per node is 1GB. Since two cores per node were active concurrently, each of the core processed 512MB of data. As a result, the task scheduling and orchestration is only just one round.

5.4.1 ModLogAnalyze

Unmodified Hadoop scales well on the 10-node configuration, with it taking only 65.63 seconds to complete the ModLogAnalyze query. The speedup provided by our new architecture is now 4.6x given that it takes 15.2 seconds to complete the same query. (Figure 5.11)

Spark performs only marginally better at 13.65 seconds (Figure 5.11). Here again, we have almost equalled Spark in a real-life interactive query. ModLogAnalyze, had not been optimized for the new architecture, and was subject to conditions similar to that of real-deployments.

5.4.2 WikipediaSearch

For the 10-node, 2 core per node WikipediaSearch, unmodified Hadoop has a JCT of 97.6 seconds. Unmodified Hadoop almost scaled linearly in this experiment. Our implementation has a cached JCT of 9.06 seconds. Speedup, for our cached run therefore is almost 11x. Spark has a runtime of 6.4 seconds. We are only 2.64s slower.

The number of rounds of scheduling and orchestration required here is 1, which explains 2.64 second gap between our runtime and Spark’s runtime.
Figure 5.11: Ten-node, Two-core 10GB input *ModLogAnalyze* JCT

Figure 5.12: Ten-core, Two-core 10GB input *WikipediaSearch* JCT
Evaluation of scaling was done using an interesting technique similar to ‘Parallel Efficiency’[11]. Since we had limited our experiments to atmost 2 cores/node; we calculated the scaling factor w.r.t the JCT (of a particular implementation) for 512MB of input data per core. We call this the normalized scaling factor.

For e.g., in the first experiment (5.2), our implementation took 13.23 seconds. But since 1024MB was the split size used, the JCT per 512 MB per core is 6.615 seconds. This was set as the baseline for our cached hadoop-run. All the other following experiments (5.3.1, 5.3.2.1, 5.4.2) were all factored for the same 512MB/core JCT values and normalized against the baseline. Figure [5.13] For perfect linear scaling, the scaling factor had to be close to 1.0 in all the experiments. As we see, both unmodified Hadoop and our implementation are within 1-1.5 range of the scaling factor.

Spark scales well overall, because of its thread-based workers, but for the one experiment with the higher input load/core (Section 5.3.2.1), the scaling factor is awry at around 2.46 (Figure [5.13]). This is the same experiment where our cached Hadoop-outperformed Spark.
5.6 Conclution

In our evaluations, our new architecture consistently matched Spark’s performance. The new architecture is 4.5x-12x faster than unmodified Hadoop on which our implementation was based on.

Our implementation was slower than Spark by less than 5% of total unmodified Hadoop JCT, which amounted to a maximum of 5-10 seconds. This was mostly observed to be caused by the higher task-scheduling and orchestration requirements of Hadoop. In one of the experiment where the amount of input load per core was high, we even outperformed Spark by 28%.

The serialized version of our implementation was also 7.8x faster than unmodified Hadoop JCT. Memory consumption – of both serialized and non-serialized version – was within limits, even though that was not an area we concentrated our efforts on. We scaled almost as well as unmodified Hadoop in this bare-bone, prototype implementation.

Most of the Hadoop Jobs that we ran in our experiments were ad-hoc, with no changes for the new architecture. Given that, our performance boost, maintaining full backward compatibility is very good. We also did not explicitly set the data-cache, or create new API’s to set them. This means that industry adoption of this architecture would be much quicker – since they will be able to see immediate benefits without having to change any code.
Chapter 6

Related Work

Hadoop, by itself, had it’s own in-memory buffering mechanism. Every map-task buffered the output intermediate Key-Value pairs in memory. These Key-Value pairs, however, are not retained over Hadoop Jobs. Moreover, the Map outputs were spilled to disk (Section 2.2.5). The readback from disk would not be as fast as an in-memory cache. Also, the key-value pairs were buffered post map operation – which is not the ideal set of key-value pairs to be cached.

6.1 Technologies based on Hadoop

Apache Hive[6], developed by Facebook in late 2009, is technically a data warehousing solution that uses Hadoop underneath. Hive supports a query language called HiveQL, which is a subset of the SQL standard. The Hive compiler translates input HiveQL to a directed acyclic graph (DAG) of MapReduce jobs. Hive uses specific file-systems (including HDFS) as input datasets.

Apache Oozie[50], is a workflow scheduler system for Hadoop. It creates a workflow or DAG of Hadoop Jobs based on the dependencies of their inputs and outputs. In an event of a complete of particular Job, the next Job within the control flow is notified and triggered.

Pig[51], is a procedural programming language and client side script that creates MapReduce Job DAG’s and Hadoop pipelines.

Apache Ambari[52] is a workflow provisioning software for Hadoop. It helps monitoring clusters of Hadoop deployments, and is used to monitor overall metrics as well. It provides a dashboard and a central management console for managing the entire cluster.

Apache Giraph, Hama are two other technologies based on Hadoop, which are explained in the upcoming section.
Apache Hadoop has two major versions which are being developed side-by-side, the 1.x branch and the 2.x branch. The 2.x branch is newer, and is more actively developed.

The 2.x branch consists of Hadoop-YARN plus MRv2 (MapReduce v2)[22] that logically separates cluster management code, resource management code (per application) and the actual application code (e.g MapReduce). It is now technically possible to run other distributed applications alongside MRv2 on the same cluster and defining hard resource limits per application. Our thesis is based on the 1.x branch of Hadoop, since the codebase is much stabler.

6.2 Recent advancements

HaLoop[10], introduced in late 2010 was the first major successful research initiative that introduce various caching mechanisms in Hadoop. Invented mostly keeping in mind iterative machine learning applications, its four major caches were used by Map and Reduce tasks. It was one of the earliest research developments that was successful in reducing Job completion times. HaLoop’s new loop-aware scheduler specifically catered to iterative machine learning algorithms and vastly reduced shuffle and network load in later iterations. Specific API’s and programming support capabilities had also been added in Hadoop MapReduce to keep up with the new set of features. HaLoop, however, was a research project and never gained any major traction in the developer community.

With Dryad, Microsoft tried to generalize MapReduce by moving to a DAG of connected phases of computation; rather than a predefined Map or Reduce. They allowed a number of user-defined elements and even permitted the data-flow between the connected phases to be rather flexible – allowing sockets, shared-memory, pipes, to name a few. They were also one of the first to introduce the concept of dynamic optimization of DAG during runtime. A very interesting paradigm that evolved of Dryad was the ability to use MapReduce at runtime using high-level language compilers such as DryadLINQ[53]. Dryad has been discontinued from active development since 2011.

Apache Mahout, is a collection of algorithms and libraries specific to machine learning algorithms, many of which use Hadoop as the core execution platform.

Pregel[54], by Google, is a graph processing architecture based on the Bulk Synchronous Parallel[5] (BSP) model of distributed computation, intended to run on clusters consisting of commodity hardware. This system is designed to be scalable, fault-tolerant and abstract in nature to support various graph processing algorithms. Apache Giraph, based on the same paper[54] as Pregel, utilizes Hadoop MapReduce to process graphs. Apache Hama is another project that uses Hadoop internally, but is able to
accept a much more generalized BSP model of computation. All the Apache projects are open sourced and being actively developed.

With Apache Hive, Ambari, Giraph, Hama, Oozie et.al Hadoop has an established ecosystem of MapReduce based software that solve generic computational problems. All of them have Hadoop’s MapReduce as its core. Any large-scale performance improvement to core Hadoop would propagate to all the systems running on top of it.

6.3 Spark & related technologies

We took a lot of inspiration from Spark[1] at every step of our work and we always tried to compare and contrast with Spark. One of the constant questions we asked was as to why Spark moved away from the whole Apache Hadoop ecosystem and re-wrote everything from scratch. In hindsight, we do now know the reasons why (enormous codebase, lack of proper resource/cluster manager at that time, assumption that in-memory cache would mean huge-internal re-architecting undertaking for Hadoop, etc.) – but we believe our solution does imbibe all the performance benefits that Spark provides, while holding onto backward compatibility and enabling easier adoption.

Spark[1], developed by AMPLab at UC Berkeley, introduced the concept of Resilient Distributed Datasets[38] – distributed in-memory datasets which are fault-tolerant and also provide a huge performance improvement when applied for iterative algorithms and interactive data analysis tools. They use the concept of lineage to track coarse transformations applied to each dataset for fault-tolerance. They believe that Spark is abstract enough to support a number of specialized programming models. A number of projects have spawned surrounding Spark, similar to the projects that have been modeled around Apache Hadoop.

- **Shark[55]**, based on top of Spark, is similar to Hive and executes declarative SQL-type queries by using a query-to-DAG and DAG-to-Logical plan compiler. Shark is able to optimize DAG mid-way during execution using PDE optimizer – which uses various input factors. This is slightly similar to DAG runtime optimization by Dryad.

- **GraphX[56]**, uses Spark as the underlying computation engine for graph processing, in a manner similar to Apache Giraph and Hama.

- **MLLib[57]** includes a number of machine learning libraries for regression, clustering, classification, etc. to be run with the Spark runtime.
Spark comes with a DryadLINQ-esque runtime environment that gives it the ability to execute lazy-actions on the fly.

Spark and our work is similar in the aspect that they both hold in-memory cached Key-Value pairs which are best suited for iterative algorithms and repetitive Jobs. Spark has evolved into a much more generic computation model that supports specialized Programming models.

Spark shows the same/similar amount of performance improvement over Hadoop as with our implementation (4.5x-12x). Spark has a fully-implemented a lineage recovery mechanism that tracks the number of coarse-transformations a particular dataset undergoes. Our implementation plans to provide that support, as noted in the Future Work section. Both Spark and our implementation showed similar amount of slowdown when using memory-efficient serialized cache as opposed to pure object cache. Lower memory footprint came at a minor cost, even while using a fast serializer-deserializer such as Kryo[31]. In our implementation, we also saw a much lower Garbage Collector kickoff when using the said serialization techniques – because the total number of Java Object stored were reduced to two per split. We have identical LRU recycling mechanisms for in-memory cache and similar checkpointing systems for disaster recovery.

A major number of observations on MapReduce as noted by Shark[55] paper were also observed while working through our implementation. Task Scheduling overhead was quite high, and task assigning/startup was slow. To reduce this negative impact, we reduced the time between heartbeats in Hadoop (Section 2.2.2). The persistent process architecture also helped counter it to some extent. By storing Key-Value pairs to in-memory cache, we lessened the impact of an inferior data layout. We also observed that “Engineering details had significant performance impact”[1] and therefore our initial study and sample implementation on the Search Job fine-tuned and profiled every possible workflow within Hadoop. (Section 4.7)

Similar to Shark, our initial implementation was also affected adversely by Garbage Collection (Section 4.5.1). We also noticed significant memory overhead due to the large number of pure Java Objects created. To counter these, we were able to move to a serialized Kryo in-memory cache which, at a minor deserialization expense, reduced GC collection and memory overhead significantly (Section 5.2.1). Shark seems to use cheap compression techniques such as dictionary encoding, run-length encoding and bit-packing[58] along with a columnar-memory store to achieve fast GC and compact memory-store. Shark uses couple of innovative techniques to achieve the ability of being able to resume partial DAG execution, and leverages on Spark’s lineage to enable parallelized recovery.
Even with JVM Primitive Caching, Garbage Collection for sizes of millions of objects takes in the order of minutes, which is inefficient. To lower the number of objects - Shark moves away from the row-storage data-structure to columnar-storage within the primitive JVM arrays. This, vastly reduces the number of objects being stored. Technically, our customized Hadoop implementation as a base for Apache Hive should be able to incorporate similar benefits and techniques as seen in Shark, but this is yet to be fully investigated and is deemed as Future Work.

6.4 Map, Reduce output caching mechanisms

HaLoop, mainly concentrated on iterative machine learning applications and had a similar ‘Mapper Input Cache’ as their cache. But HaLoop stored this cache, as well as two other different ‘Reduce Input’ and ‘Reducer Output’ Cache at the TaskTracker. The Mapper/Child processes were not persistent, and the cache was not in-memory. We believe that data-cache transfer to the later iteration Mapper tasks were over the network, which explains the low 1.85x speedup observed by HaLoop.

Since HaLoop was particularly invested in solving machine-learning problems, it introduced the extra caching support for faster fixpoint evaluation using reducer output cache – rather than having a dedicated MapReduce Job after every iteration. HaLoop also introduces a number of new Hadoop API to developers, which have to be set properly before the performance benefits kick in. This is unlike our implementation where we ensure no changes for the developer code. HaLoop’s Loop-aware task-scheduler is slightly similar to the delay-scheduler that we implemented. It’s task-assignment is also done with the knowledge of the TaskTrackers-to-cache mapping.

Twister, in a manner similar to HaLoop, redefines a new iterative MapReduce runtime with new caches and API specifically introduced for those type of iterative problems. They provide an in-memory cache within one JVM process daemon of Twister – which spawns the Map/Reduce worker tasks. Their publish-subscribe messaging model is rather different from what Hadoop traditionally uses.

In spite of this, their ‘Parallel Efficiency’ is close to 0.7 and for plain Hadoop is 0.6. This result of theirs on the SW-G distance calculation, which is similar to MapReduce’s own Grep/WordCount applications, only shows how well the system scales with the number of cores/computation units added. An ideal scaling would have the ‘Parallel Efficiency’ as exactly 1.0. Their experiments on an iterative application such as PageRank does not compare with traditional Hadoop, nor does it compare the
1st iteration vs rest of their iterations, so we cannot perceive how well their system performs when the in-memory cache is actually put to use.

### 6.5 Distributed fine-grained caching

**RAMCloud**\(^{[59]}\) is a distributed shared-memory storage system which operates at a much lower-level and provides the capability of fine grained-updates to table-cells. The writes and resiliency methods are quite expensive, and even though reads in RAMCloud-based-HDFS\(^{[59]}\) show 6x improvements over plain HDFS reads, this will not scale well when actually incorporated with Hadoop/MapReduce where reads would be lower than 512MB per Map Task.

**PACMan**\(^{[60]}\) memory cache was on top of HDFS and just cache’s the input data of jobs rather than the post-parsed Key-Value pairs that we do in our implementation. They have concentrated on improving the cache-eviction policy based on a all-or-nothing strategy which co-ordinates cache management such that all the input is available in-cache for a set of Jobs running in the same wave such that the overall completion time is reduce. They have been only able to simulate a speedup of 53% which is not high. Their **LIFE** cache-eviction policy is quite innovative, and is supplementary to what we’ve done. If we replace our place LRU-HDFS-backed cache with this policy and integrate it with Oozie as part of our future work, we might be able to see better completion times for Pipelined Jobs.

### 6.6 Other work

**MRShare**\(^{[61]}\) reduces the number of MapReduce Jobs by finding redundant queries/operations on the same dataset in a single batch of submitted queries to the MapReduce data-analytics platform. The system has come up with a cost model to find the ideal plan to merge the queries of a single submitted batch. Hence, the goal of MRShare is to reduce the number of redundant operations in a single MapReduce workflow, irrespective of whether its iterative/repetitive. This is quite different from what we’ve pursued, as it does not leverage from previously completed Jobs, nor does this have anything to do with in-memory caching.

**ReStore**\(^{[62]}\) stores the intermediate results that Hadoop traditionally doesn’t retain and
reuses them for future pipelines. It rewrites later workflows based on heuristic matching of previously run sub-jobs and their intermediate results. ReStore operates on a higher data-flow level, and is mainly implemented for PIG\cite{51} by using a modified JobControl-Compiler\cite{51}. It is able to enable re-use of said intermediate results at both sub-job and of whole-jobs. The output for ReStore is stored in HDFS, which turns out to be much slower. In a way, our implementation and ReStore would be complementary to each other, by leveraging on our persistent-process in-memory cache and using ReStore’s dynamic rewrite of workflows, we could have a system that in a manner similar to Shark, would be able to optimize a query mid-way.

MixApart\cite{2} enables decoupled data analytics, trying to lower cost of hardware by using disk-based caches and intelligent cache-placement aware schedulers and proactive background data-transfer mechanisms. As mentioned in the paper\cite{2} itself, any in-memory caching system – ours included – could be layered to be used on top of MixApart for improved efficiency.

Apache Gora\cite{63}, is a very new and upcoming technology which does object-to-relational mapping, using an in-memory store. Their main idea is to act as an in-memory store, similar to our implementation. Apache Gora persists the in-memory data back to column stores (HBase, Cassandra), key-value stores (Redis, MongoDB, etc) or even SQL databases and flat filesystem. It has other indexing, data-access features on top of the in-memory store.

\section*{6.7 Conclusion}

By late 2000s, there was good ecosystem of softwares that were built around Hadoop. These softwares performed non-generic parallel computations and harnessed Hadoop as their main execution engine.

By around that time, there were a number of efforts for intelligent caching since Hadoop became mainstay. For significant speedups, caching had to be done at the right place (within the worker processes/threads), and needed to be in-memory. The caching also had to be of the right set of \(<K, V>\) pairs – ideally of only the parsed input data which were to be fed as input to \texttt{map}.

Network-access of cached \(<K, V>\) pairs, or distributed fine-grained updates to them, or IPC access to cached \(<K, V>\) pairs had to be avoided. To ensure easier adoption
by industry and full backward compatibility with existing Hadoop Jobs/applications, no
new API’s could be added or programming models introduced. Many of the research
efforts such as HaLoop, Twister, RAMCloud, PACMan, etc. did not fulfill all of the
above requirements and therefore either lacked in performance, or did not enable back-
ward compatibility/adoptions.

Spark, the latest in-memory MapReduce architecture, did caching right. With thread-
based workers, it was very performant and resulted in significant speedup. Spark built
the MapReduce paradigm ground up and had its own new programming model. Porting
effort was deemed costly. Jobs had to explicitly specify the $<K, V>$ pairs that needed
to be cached.
To move from Hadoop to Spark, existing code had to be re-written in Scala and in-
memory caches had to be defined with the new APIs. Hadoop workflow of Jobs and
pipes had to be rewritten. Certain amount of re-architecting and re-deployment of clus-
ters was also necessary. Also, Spark did not have an ecosystem of softwares as mature
as Hadoop.

Our goal was to maintain the very familiar Hadoop API, but provide similar speedups
and performance boosts as observed by Spark, without any changes to application code
whatsoever.
Chapter 7

Conclusion & Future Work

7.1 Conclusion

We introduced a new in-memory cached architecture of Hadoop – that matched and in some cases outperformed Spark, which is the state-of-the-art high-speed in-memory MapReduce architecture with caching. Spark reimplemented the MapReduce paradigm from ground up. It introduced a new programming model, API’s and abstraction – requiring a very costly porting effort. We maintain the familiar Hadoop framework and API’s, thus complete backward compatibility for any existing Hadoop-based applications and analytic tools.

Our new architecture hinges on three basic improvements that we concluded from the background study – the parsing benefit, the in-memory read benefit, persistent process/reuse benefit. We implemented a very generic architecture which would run on almost on any Generic Java Key-Value pair - to almost immediate performance boosts.

Our implementation performed almost as good as Spark in many of our experiments - lagging at most by 5-10 seconds - mostly because Hadoop has higher task-scheduling and orchestration requirements. This lag is less than 5% of total unmodified Hadoop execution time. In the search experiment where the input-size of data is large we outperformed Spark by 28.9%. In a 10-node run of compute-intensive, shuffle-heavy Job, we were less than 2 seconds slower than the Spark runtime.

In our earlier background study, we started off a series of extensive experiments to understand Hadoop Jobs behaviour with an in-memory file system such as Ramfs (on top of HDFS). In an initial set of compute-intensive and disk-intensive Hadoop Jobs, we saw huge improvements when deploying the Jobs over Ramfs-HDFS. The performance benefits, varying on input sizes and compute-intensity – was about 2.5x.

In the next iteration of experiments, we started with the most pessimistic baseline –
a compute intensive Job reading a ∼135GB input of Key-Value SequenceFile pairs and comparing Job timings when run over disk vs when run over Ramfs. We argued that if we’re able to catch up with the same degree of improvement as pure 135GB of Read from disk vs Ramfs – then our Hadoop Jobs would be as performant in our new proposed architecture.

Over every iteration of the experiments, we were able to narrow down on the performance hogs by profiling. This helped cultivate a better understanding of where Hadoop was inefficient at. In our final experiment - our Ramfs deployment was able to achieve 38% of the maximum theoretical improvement over the disk-based deployment of Hadoop.

With this positive result, we went ahead with our implementation of the in-memory cached Hadoop architecture and a cache-aware scheduler. We easily saw a 11x performance boost on many evaluations over unmodified Hadoop. Even in the most compute heavy applications, our implementation was atleast 1.9x faster. As mentioned before, our prototype implementation of this architecture performed comparably to Spark in all experiments. Future work, concentrating on improving the prototype and feature-set of this new architecture seems very promising.

7.2 Future work

i) Technical improvements in prototype implementation Some of the enhancements that could be done to our prototype implementation are almost immediately obvious:

- **Thread-based workers** – Our current implementation stores the in-memory cache at the process-level, within a Mapper instance. If we move to a multi-threaded model, we could do-away with multiple-process per slave. Thread spawning is very cheap and fast, allowing us to increase scaling factor very efficiently. The in-memory data-structure of the master persistent process would be shared across the threads, in a lock-free read-only fashion for the cached runs.

- **Better scaling** – The current implementation, because of certain technical reasons, does not scale to more than 4 cores per slave. This is partly because of the multiple-tasktracker (Section 4.3) approach we took for the purposes of this quick implementation. If we move to a single-tasktracker multi-threaded method, we should be able to scale easily.

- **Split management** – Another possible technical quirk we could rectify is
better management of splits. Currently, for our experiments, the configuration
is manually set. Ideally, we’d like the JobTracker to automatically create larger
split sizes and automagically handle variable split sizes in accordance with the
cache stored in the Mapper. In a multi-threaded implementation, automatic
split size management would be a careful decision involving the number of
cores, the cache size and other factors (Fig. 2.3).

ii) **Fallback and Lineage** – The current fallback mechanism is only a HDFS-backed
propagation of cache-objects. This is not efficient and is slow.

An ideal methodology would be to have an intelligent split-mechanism, as dis-
cussed above, along with a better lineage-type[38] structure tracked by the Job-
Tracker. This should be exposed to the system managing the overall pipeline/DAG
of Hadoop Jobs. Failures of slaves or data-loss of the cache-objects would now re-
sult in re-running the affected splits over the same set of operations or Hadoop Jobs
tracked by the structure. Fault-tolerance, achieved by this method, would need to be
evaluated against the current HDFSBacked-LRUcache (Section 4.5.2) method.

iii) **Event-driven architecture** – Hadoop’s architecture is based on heartbeats, which
is almost a polling based mechanism (Section 2.2.2). This has resulted in a number
of task-scheduling related inefficiencies that we saw in the evaluations. Replacing
this underlying heartbeat architecture with a much more responsive event-driven
message-passing library would be ideal. We would almost surely notice an increased
performance in experiments that had lower split sizes, which required larger number
of tasks to be scheduled.

iv) **In-memory Map output** – We could store the Map output (Section 2.2.5) directly
onto memory, rather than to local slave disk. Depending on the type of Job, we
may benefit if the intermediate <K, V> pairs are fetched by the Reducer using the
in-memory store. The current methodology of HTTP get of Map output by the
Reducer could be a bottleneck.

v) **Zero-copy serialization** – Instead of relying of Kryo[31], it will be nice to integrate
zero-copy mmap() based serialization formats such as Google’s FlatBuffer[64] and
other offerings like Cap’n Proto[65]. Deserialization costs will be zero, while it
might require a previously defined schema for the <K, V> pair format. This needs
to be investigated.

It will be interesting to apply this new implementation of Hadoop as the underlying
engine to Hive and compare it against Shark. This would enable very quick repeat-
query processing for Hive. Hadoop pipelining and systems such as Apache Oozie[50] would also be interesting use cases of our implementation. Though not an immediate afterthought, it’s thought provoking to see how this could manifest into a generic system for machine learning[9], iterative algorithms, graph processing[7], BSP[8] and other non-generic MapReduce computation problems.
Appendices
The following source code was used as a standalone single-process single-core Java application that mimicked the in-memory cached search functionality of WikipediaSearch. This was used for the stress-test as explained in Section 4.7.

```java
String line;
int count = 0;
StringTokenizer tokenizer;
try {
    BufferedReader br = new BufferedReader(new FileReader(file));
    ArrayList<SimpleEntry<Object, Object>> list = new ArrayList<SimpleEntry<Object, Object>>() {
        while ((line = br.readLine()) != null) {
            tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                list.add(new SimpleEntry<>(new String(tokenizer.nextToken()), new Long(0)));
            }
        }
    }
    cache.put(identifier, list);
} catch (FileNotFoundException e) {
    System.out.println("fnfexception");
} catch (IOException e) {
    System.out.println("ioexception");
}
System.out.println("Loaded to Memory"); //--- END PHASE 1 ---//
System.gc(); // Forced Garbage Collect before Phase 2
//--- PHASE 2 ---//
// Search phase
System.out.println("GC Done");
int iter_list = 0;
long starttime = System.currentTimeMillis();
ArrayList<SimpleEntry<Object, Object>> tempref = cache.get(
```
identifier);
    int length = tempref.size();
    while(iter_list < length) {
        if(map((String) tempref.get(iter_list).getKey(), (Long) tempref.get(iter_list).getValue())) {
            count++;
        }
        iter_list++;
    }
    System.out.println("indiat" + count + " time:" + (System.currentTimeMillis()-starttime));
    //................................................................................ END PHASE 2 ........................................

    static boolean map(String key, Long value) {
        return key.equalsIgnoreCase(search_term);
    }
}
Appendix B

Even though the initial experiments of Section 3.2.1.2 succeeded, we thought that the input SequenceFile’s were a bit too small for a 100% confirmation towards the proposed new architecture. Therefore, in the next set of experiments, we tried to scale the ModLogAnalyze Job over 1TB of input data.

In this appendix we will go into detail some of the scaled experiments we conducted before finally succeeding in the Modified SequenceFileFormat experiment in Section 3.2.2.

B.1 Scaled SequenceFile experiments with variable split-size

These set of experiments were conducted one after another with the intention to further promulgate the difference in execution time between SequenceFile-Hard-drive and SequenceFile-Ramdisk input sources, even under stringent, computationally heavy conditions.

Every experiment was averaged over 3 runs, studied and then based on it further parameters were tweaked for the next set of experiments. A summary of the experiments run are given below. All of them have been tabulated and graphed in [B.1],[B.1].

The experiments were run in a single-machine Hadoop installation using a ‘Fat’ node (5.1), which possessed 512GB of RAM and therefore helped create large Ramdisks to accommodate the in-memory SequenceFiles.

Our scaled SequenceFiles experiments were pessimistic in nature. We took a compute heavy, shuffle-intensive ModLogAnalyze Job, stripped it of its parsing benefit (SequenceFile input), deliberately used smaller input split sizes (Figure 2.3), hinged only on the in-memory benefit (from Ramdisk) – and checked if we could eek out any performance improvements in the experiments.
Table B.1: JCT, Number of Splits/IF with SequenceFile on Hard-drive (HD) or Ramdisk

<table>
<thead>
<tr>
<th></th>
<th>RAM Disk JCT (sec)</th>
<th>RAM Disk</th>
<th>RAM Disk</th>
<th>RAM Disk</th>
<th>RAM Disk</th>
<th>RAM Disk</th>
<th>RAM Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default Split (135MB)</td>
<td>5350</td>
<td>4625</td>
<td>4643</td>
<td>4166</td>
<td>2480</td>
<td>2424</td>
</tr>
<tr>
<td></td>
<td>1 GB Split (135MB)</td>
<td>4630</td>
<td>4648</td>
<td>4653</td>
<td>4212</td>
<td>2501</td>
<td>2415</td>
</tr>
<tr>
<td></td>
<td>2 GB Split (135MB)</td>
<td>4640</td>
<td>4658</td>
<td>4663</td>
<td>4212</td>
<td>2501</td>
<td>2425</td>
</tr>
<tr>
<td></td>
<td>1 GB Split (1350MB)</td>
<td>4640</td>
<td>4658</td>
<td>4663</td>
<td>4212</td>
<td>2501</td>
<td>2425</td>
</tr>
<tr>
<td></td>
<td>2 GB Split (1350MB)</td>
<td>4640</td>
<td>4658</td>
<td>4663</td>
<td>4212</td>
<td>2501</td>
<td>2425</td>
</tr>
<tr>
<td></td>
<td>1 GB Split (1350MB), no Reduce</td>
<td>4640</td>
<td>4658</td>
<td>4663</td>
<td>4212</td>
<td>2501</td>
<td>2425</td>
</tr>
<tr>
<td></td>
<td>2 GB Split (1350MB), no Reduce</td>
<td>4640</td>
<td>4658</td>
<td>4663</td>
<td>4212</td>
<td>2501</td>
<td>2425</td>
</tr>
</tbody>
</table>

Figure B.1: JCT with HD/Ramdisk as SequenceFile storage; for diff. ‘Split Size (Input File size), Reduce/No Reduce’
B.1.1 Default-split, (135MB Input File [IF]) experiment

Prior to running this experiment, 1TB (1024x1GB files) of raw HTTP logs were taken from the EECG (http://www.eecg.toronto.edu) servers and Jobs were run to produce 1024x135MB (∼135GB) SequenceFiles.

Hadoop configuration parameters were not modified and were exactly the same as Basic SequenceFile Experiment (Section 3.2.1.2). This means that the InputSplits were 64MB (default-split) each. So for a 135MB Input File (IF), 3 splits per file (64MB + 64MB + 7MB) gave a total of 3072 (1024x3) InputSplits and 3072 Map tasks.

We notice that for all the 3072 Map tasks and 1 Reduce task (of the Job) to finish for ModLogAnalyze, it takes approximately 5350-5370 seconds irrespective of whether the SequenceFile is read from Hard-drive or Ramdisk. Simple calculation from the read-rate experiment shows that for an input for 135GB, the difference in read-time is at least 750 seconds, which is not what we observe here.

We could later reason this observation because of one sole reason: 2048 of the 3072 Map tasks were reading 64MB each, which for both Hard-drive or Ramdisk takes an insignificant amount of time to be read (∼ 0.05 sec). It is even more insignificant for other 1024 tasks which were reading 7MB each. Therefore, we had to increase the InputSplit size and reduce the number of Mapper tasks to enforce larger reads in the Job.

B.1.2 1GB Split, (135MB IF) and 2GB Split, (135MB IF) experiments

The Split size is calculated in Hadoop by the formula:

\[ \text{split} = \max(\text{mapred.min.split.size}, \min(\text{mapred.max.split.size}, \text{dfs.block.size})) \]  (B.1)

where

- \text{mapred.min.split.size} – defaults to 64MB, but is set at 1GB/2GB in this subset of experiments.
- \text{mapred.max.split.size} – defaults to 64MB.
- \text{dfs.block.size} – defaults to 64MB for HDFS and is generally considered a global value. Changing this value affects only the newly created files for existing deployments of Hadoop, as is the case we are dealing with, and therefore not ideal.
The split value is now 1GB, but since the minimum number of Map tasks spawned is ≥ to the number of input files\(^1\), we have 1024 Map tasks reading 135MB input each. Setting mapred.min.split.size to 2GB also has exactly the same end result. The number of splits, Map tasks and input files are the same 1024.

We notice that execution time for 1024 Map tasks and 1 Reduce task for this Job is more or less the same with either the SequenceFile stored on Hard-drive or Ramdisk (Table [B.1]). Through the read-rate experiment, we note that the difference in time-taken to read 135MB from the Hard-drive vs Ramdisk is about 0.8 seconds, which is not significant for a Map task that takes ∼4-5 seconds. For 1000+ Map Tasks, in this high-latency system, the overall execution time difference is amortized. We had to make the disk-reads even larger.

### B.1.3 1GB Split (1350MB IF) Experiment

For this experiment, the 1TB (1024x1GB) raw input files were re-processed to create 102x1350MB SequenceFiles (an approximation). These 102 files were used as input.

Ten 135MB SequenceFiles could not be concatenated with each other to form the 1350MB file because the SequenceFileFormat is binary and therefore had to be re-processed again.

The split size using equation (B.1) still remains 1GB. The number of input files is reduced to 102. The split size is lower than the IF size, hence the number of Map Tasks spawned for each IF is 2 (1024MB + 326MB Map Task).

Assuming little or no benefit from the 326MB Map task by reading from Ramdisk, the other 102 Tasks each reading 1GB should provide a conservative benefit of :-

\[
102 \times 6 \text{ (second per GB according to read-rate experiment (3.2.1.1))} \approx 600 \text{ seconds}
\]

But we observe only an improvement of about 46 seconds (Table B.1), therefore something’s amiss.

Profiling the Hadoop Job (using HPOF[25]), we see that each of the Map tasks (either HD or Ramdisk) were taking almost exactly the same time for execution because of a Combiner thread. i.e. for large amount of intermediate key-value data (spill [48],[2.2.5]), Hadoop proactively starts a local Reduce-type thread within the Map task, to locally Combine the <K, V> pairs – and this takes up a significant portion of the CPU execution/total execution time as seen in Appendix [C.4] (org.apache.hadoop.mapred).

\(^1\conf.setNumMapTasks()\) can be only used to hint at a higher number of Map tasks to be run than the number of input files
Appendix B.

MapTask$MapOutputBuffer.sortAndSpill, 23%). Therefore, for the next set of experiments, the Reduce and Combiner part will need to be removed from interfering.

It could not be determined at this stage if the total read-time from the disk(s) for the Map tasks were equal.

B.1.4 1GB and 2GB Split (1350MB IF) No Reduce experiments

In this set of experiments, we removed the Reduce and Combiner of ModLogAnalyze and set the OutputFormat[66] as NullOutputFormat.

Shuffle and sort operations still took place. This effectively reduced ModLogAnalyze to an identity Job – one which just reads the $<$K, V$>$ pairs from the SequenceFile. In the 2GB split experiment, the number of splits (and hence the number of Map tasks) is 102.

We see from Table [B.1] that by removing the Reduce part, the execution time decreases by more than 1600 seconds, going to say that that our conclusion from the previous experiment was correct. However, the total execution time for the Job hovers between 2400-2500 second, for both 1GB and 2GB split. The difference in execution time between Hard-drive and Ramdisk is still negligible. Java HPROF profiling was done to give more insight into the anomaly. Appendices [C.5], [C.6] present the dot-graph[67] of the approximate percentage of CPU execution time for each function within a Map task.

Since sampling (HPROF) was used to profile the Jobs (the other times option being unfeasible here), we’ll notice that the percentages do not add up properly. The org.apache.hadoop.mapred.Child$.run function is the starting point of all Map tasks and should ideally add to 100%. So, any sub-function which is derived as a part of org.apache.hadoop.mapred.Child$.run, should be calculated relative to it.

The most important functions to be kept track of are:

- org.apache.hadoop.mapred.Child$.run – This is the main point of entry for a Map task being run.


- org.apache.hadoop.hdfs.DFSClient$DFSInputStream.read – The HDFS read time. This is quite important; in case of Hard-drive we expect the percentage to be much higher than that of Ramdisk.
• Any other extraneous function that uses up CPU %. The appendices [C.5], [C.6] were carefully chosen over multiple profiled tasks, since it resembled to what we thought was the closest average graph. This again is because of the approximate sampling method used for HPROF. There is also inherent randomness in the graph – haphazard placement of edges/nodes when using dot[67, 27].

The observations are these:

i) For the average Hard-drive execution, we notice that 40-45% of the execution time is spent on DFSInputStream.read and about 15% of the time is spent on the deserialization code.

ii) For the average Ramdisk execution, we again see that 40-45% of the execution time is spent on DFSInputStream.read and about 15% of the time is spent on the deserialization code.

iii) The org.apache.hadoop.mapred.Child$.run takes about 64% of the execution time. This means that, more than 70% (45/64) of the time is spent on disk read and about 25% of the time is spent for deserialization in both the cases.

From this, we believe that there is something internal to the way SequenceFileInputFormat’s RecordReader operates that makes it inefficient to read from Ramdisk. It is possible that the RecordReader of SequenceFileInputFormat reads the InputStream line-by-line, which is not conducive to Ramdisk-based reads. This still needs to be investigated.

Our next step was to modify SequenceFileInputFormat such that it optimizes its read operation for Ramdisk; and then re-run the last set of experiments to see if we notice any improvement.

B.2 Memcached experiments

Before the SequenceFile experiments, we ran a couple of experiments quickly integrating Memcached[39] with Hadoop. Memcached is a generic distributed based caching system. Memcached stores large number of <K, V> in-memory in a distributed hash table.

Memcached was used to store the <K, V> pairs (akin to SequenceFile on Ramdisk) and retrieved later in subsequent runs. This took advantage of the parsing benefit and in-memory benefit.

RegexRecordReader parsed the input HTTP log file and subsequent RegexRecordReader instances check if the particular input-file split was pre-parsed already – and retrieves the stored <K, V> pairs from Memcached.
RegexRecordReader code (part of ModLogAnalyze’s InputFormat) was modified to make this as application independent as possible. It bore the brunt of \( <K, V> \) store/retrieve logic. Since memcached-solution was not our final intended implementation, the integration was quick and hacky.

**Notes and experimental observations:**

i) Input file-split indexes were used as reference points to check if the distinct group of memcached-entries were already cached or not. Hadoop Jobs outputs from the memcached-code were verified to be correct.

ii) Memcached had two modes, *async* and *sync*. Async mode did not guarantee an immediate update to memory values. Async mode is eventually consistent and therefore much faster.

iii) Using 1GB raw Hard-drive input file and a regular expression heavy *ModLogAnalyze Job*.

<table>
<thead>
<tr>
<th></th>
<th>Cached run</th>
<th></th>
<th>Non-cached run</th>
<th></th>
<th>Cache-build run</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>With</td>
<td></td>
<td>Sync</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>memcached-library</td>
<td></td>
<td>Async</td>
</tr>
<tr>
<td>JCT (sec)</td>
<td>270</td>
<td>250</td>
<td>85</td>
<td>69</td>
<td>320</td>
</tr>
<tr>
<td>memcached-library loaded</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcached-read operation</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>memcached-write operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>disk-read operation</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table B.2: Properties and runtime of various memcached-based experiments

To understand this unexpected result, we profiled some of the Jobs, analyzed the HDFS-file-read counters and collected Memcached’s *cmd_set* and *cmd_get* statistics. We understood that:-

i) Memcached does not have pure shared-memory read access. All (async/sync) reads are over the network (even if the data is cached local to the node) and is slow.

ii) Memcached write is faster in async than sync – since it just has to locally persist in memory and propagate to other Memcached machines later over the network.

iii) Read of memcached-values in sync mode is even slower. Memcached does a poor job of single-node-instance memory-object access since the read/writes are over the network.
It is good to note that Memcached+Hadoop integration doesn’t perform well as the SequenceFile+Ramdisk implementation. In retrospect, we should’ve collected all the $<K, V>$ pairs and then serialized them to minimize the number of API calls. Either way, this experiment was not very successful because we could not ascertain if parsing benefit had any impact on Job completion times.
Appendix C

The following call-graphs of various experimental runs (referenced in this thesis) were made using Java HPOF[25] and dot[67, 27]. All of them are in high-resolution and can be zoomed in when viewed as a PDF.
Figure C.1: Java profile graph of a Non-cached run [High-resolution]
Figure C.2: Profile of a Cached run
Figure C.3: Profile of the 1024MB Map task reading SequenceFile from Hard-drive
Figure C.4: Profile of the 1024MB Map task reading SequenceFile from Ramdisk
Figure C.5: Profile of the 1350MB Map task reading SequenceFile from Hard-drive, devoid of any Reduce/Combiner
Appendix C.

Figure C.6: Profile of the 1350MB Map task reading SequenceFile from Ramdisk, devoid of any Reduce/Combiner.
Bibliography


