Migration of batch log-processing job from apache hadoop map-reduce to apache spark in the public cloud

Leonid Vasilyev

Submitted as part of the requirements for the degree
of MSc in Cloud Computing
at the School of Computing,
National College of Ireland
Dublin, Ireland.

September 2015

Supervisor Michael Bradford

Submission of Thesis and Dissertation

National College of Ireland

Research Students Declaration Form

(Thesis/Author Declaration Form)

Name: _____

Student Number: _____

Degree for which thesis is submitted: _____

Material submitted for award

(a) I declare that the work has been composed by myself.

(b) I declare that all verbatim extracts contained in the thesis have been distinguished by quotation marks and the sources of information specifically acknowledged.

(c) My thesis will be included in electronic format in the College Institutional Repository TRAP (thesis reports and projects)

(d) ***Either*** *I declare that no material contained in the thesis has been used in any other submission for an academic award.

Or *I declare that the following material contained in the thesis formed part of a submission for the award of

_____

(State the award and the awarding body and list the material below)

Signature of research student: _____

Date: _____

Submission of Thesis to Norma Smurfit Library, National College of Ireland

Student name: _____ Student number: _____

School: _____ Course: _____

Degree to be awarded: _____

Title of Thesis: _____

_____

_____

One hard bound copy of your thesis will be lodged in the Norma Smurfit Library and will be available for consultation. The electronic copy will be accessible in TRAP (http://trap.ncirl.ie/), the National College of Ireland's Institutional Repository. In accordance with normal academic library practice all theses lodged in the National College of Ireland Institutional Repository (TRAP) are made available on open access.

I agree to a hard bound copy of my thesis being available for consultation in the library. I also agree to an electronic copy of my thesis being made publicly available on the National College of Ireland's Institutional Repository TRAP.

Signature of Candidate: _____

For completion by the School:
The aforementioned thesis was received by_____ Date:_____

This signed form must be appended to all hard bound and electronic copies of your thesis submitted to your school

Abstract

With the increasing adoption of cloud-based infrastructure the problem of efficient utilization of provisioned resources becomes more important, since even in a *pay-as-you-go* model computing resources are allocated and charged in a coarse grained way (e.g. the whole virtual machine per hour). This problem becomes major in the batch processing systems, where computational resources are organized into a cluster. Even small optimization of applications running on such systems can result in significant cost savings.

In this thesis we evaluate one such application — *JournalProcessor*, which is a batch log-processing job that aggregates and indexes logs containing metrics data. *JournalProcessor* application itself built using *Apache Hadoop MapReduce* engine that runs on top of *Apache Hadoop YARN* cluster resource manager, which is deployed onto *Amazon EC2* public cloud using *Amazon Elastic Map Reduce (EMR)*.

The research question of this thesis is the following — is it possible to migrate *JournalProcessor* from *Apache Hadoop MapReduce* data processing engine to more general data-stream oriented system – *Apache Spark*? Our hypothesis is that by migrating the application to *Spark*, the utilization of provisioned cluster resources will increase, and the running time of the job will decrease.

Our contributions include: description of the application to generate the workload (*JournalProcessor*), generic methodology for migrating *MapReduce* applications to *Spark* and the detailed evaluation of metrics produced by a cluster.

Dedicated to my wife Maria. For all her help and support.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

MapReduce (Dean & Ghemawat 2004) dominates in the industry as an approach for solving *Big Data* (Laney 2001) problems. Yet in the recent years, industry realized that not all *Big Data* problems can be solved efficiently with *MapReduce* — for example algorithms that require multiple iterations over the same dataset, such as *Machine-Learning* (Zaharia et al. 2010). As a result, several successor systems were proposed, such as *Apache Spark* (Zaharia et al. 2010) and *Pregel* (Malewicz et al. 2010). These systems offer a superset of *MapReduce* functionality: a *Direct Acyclic Graph (DAG)* of transformations over the input data set (Isard et al. 2007). *Spark* extends it even further by allowing cycles in the graph.

A problem does arise, however. How one approaches a migration of existing applications from *MapReduce* engine into *Spark* engine without violating the correctness and performance?

The research question of this thesis is the following — is it possible to migrate *JournalProcessor* from *Apache Hadoop MapReduce* data processing engine to more general data-stream oriented system – *Apache Spark*? Our hypothesis is that by migrating the application to *Spark*, the utilization of provisioned cluster resources will increase, and the running time of the job will decrease.

The solution we propose is to build a methodology which provides a process for mapping between functionality of *MapReduce* and *Spark*. This methodology should not be a simple one to one transition, but instead utilized the provided API efficiently.

The scope of the thesis is evaluation of two systems: *Apache Hadoop* which is the most used implementation of *MapReduce* (Dean & Ghemawat 2004) engine and *Apache Spark* (Zaharia et al. 2010) which is developed as a successor of *Hadoop's MapReduce*.

We have picked a typical *MapReduce* job — *JournalProcessor*, that is used for a few years in large enterprise environment. We then migrate the job to *Spark* engine, and validate that both jobs produce the same results.

Both systems are deployed on top of *Amazon EC2* public cloud, and managed by *Amazon Elastic MapReduce(EMR)*. Cluster resources are allocated via cluster management system *Hadoop YARN* (Vavilapalli et al. 2013). Input and output data is stored in *Amazon S3*.

During the evaluation we collected performance metrics using *Ganglia Monitoring System* (Massie et al. 2004) and *AWS EC2* instance metrics provided by *AWS CloudWatch*.

Out of Scope

Both jobs were evaluated using the same hardware configuration. Evaluation various types of hardware is out of scope for this work, but could be addressed as a future work. Although, the configuration of both MapReduce and Spark jobs was changed to archive maximum performance, we do not conduct exhaustive evaluation of available configuration options.

1.0.1 Summary

We were not able fully confirm out hypothesis during the experiment. Despite the we were able to archive significant improvement in cluster utilization with *Spark*, the running time of a *MapReduce* solution was 65% less even with less efficient utilization of cluster resources. We found that the performance bottleneck is *SSD I/O* throughput.

1.0.2 Outline

The rest of the thesis is organized as follows. Chapter 2 presents the survey of existing work in the area of alternatives to *Hadoop MapReduce*, as well as different approaches to optimization of existing applications. We also survey alternatives to *Apache YARN* since we believe that cluster management system is crucial for performance. Chapter 3 presents the design of our approach to migration and evaluation of results. It also presents the methodology for migration. In Chapter 4 we describe the implementation of our solution. Chapter 5 presents the evaluation of our results using various metrics. Chapter 6 summarizes our findings and present pattern we observed during implementation. The full source code of a *Spark* solution can be found in Appendix C.

Chapter 2

Literature Review

In this chapter we look at related work that has been described in the literature. Section 2.1 presents the survey of *Big-Data* processing systems. In section 2.2 we survey key components of *Cluster Management Systems*. We discuss the performance of *Spark* and cluster configuration in subsection 2.1.3 and 2.1.4 respectively.

2.1 Survey of Big-Data Processing Systems

2.1.1 Processing Models

In the original *MapReduce* paper Dean & Ghemawat (2004) established the interface for parallelization: `map(k1, v1)` $\mapsto$ `list(k2, v2)` and `reduce(k2, list(v2))` $\mapsto$ `list(v2)`. This interface saw wide adoption by the industry as a de-facto approach for solving *Big-Data* (Laney 2001) problems. Dean & Ghemawat (2004) developed few optimizations to the MapReduce model: custom partitioning logic, ordering guarantee in the reducer, combiner function and backup tasks. As a limitations of MapReduce Dean & Ghemawat (2004) stated that MapReduce only supports acyclic data flows, and lack of atomic operations.

Isard et al. (2007) generalized the MapReduce interface proposed by Dean & Ghemawat (2004) into distributed data-parallel application. He claimed that most general abstraction for these applications in *Direct Acyclic Graph (DAG)*. Another improvement introduced by *Dryad* (Isard et al. 2007) was separation of operations on data from middleware that provides parallelization. Dryad also introduces declarative query language which abstracts the construction of transformations.

The *DAG* model was extended even further by *Spark* (Zaharia et al. 2010) and later by *Naiad* (Murray et al. 2013). Both Spark and Naiad support cyclic computations and, unlike MapReduce (Dean & Ghemawat 2004) and Dryad (Isard et al. 2007), low latency queries by relying primarily on main memory(DRAM) as a main storage media.

The key component of Spark (Zaharia et al. 2010) is an abstraction over distributed shared memory: *RDD* (Zaharia et al. 2012). It allows to archive scalability and fault tolerance of MapReduce. Important characteristic introduced by Zaharia et al. (2012) is the *lineage* which is used instead of replication to recover from a partial data-loss in the system. Different approach was taken by *Pregel* (Malewicz et al. 2010), which is not data-flow system, instead it is a message passing system that keeps local state in memory, and applies modification to its local state instead of passing data over the network.

2.1.2 MapReduce Performance

Blanas et al. (2010) stated that major issue with MapReduce model is lack of support for *join-like* operations, i.e. MapReduce is not designed to combine data efficiently from multiple sources. *Spark* (Zaharia et al. 2010) addresses this with broadcast-based joins. Another approach was seen in Logothetis et al. (2011) in order to optimize the problem of distributed aggregation. Logothetis et al. (2011) proposed to perform the map phase on the nodes where data has been generated and dedicated nodes are running the reducers. Another novel approach that Logothetis et al. (2011) proposed is to let the user control the fidelity of the data.

2.1.3 Spark Performance

Davidson & Or (2013) observed that shuffle between two stages is the major source of performance issues in *Spark*. They found that *Spark* utilizes underlying file system in sub-optimal way: *Spark* produces a lot of small files which cause random I/O versus sequential. Also, authors propose to use *Ext4* file system instead of *Ext3* to allow better write performance.

Another area of improvement found by Li et al. (2014) is a *HDFS* (Borthakur 2007) performance. Their proposed system *Tachyon* uses memory as a primary storage. Experiments report 110-x improvement over *HDFS* on write operations. Li et al. (2014) also observed that there is a hierarchy of storage: main memory has 10-100 GB/sec bandwidth, datacenter network has 1.25 GB/sec and SSD disks has 1-4 GB/sec.

Tachyon does not take advantage of this hierarchy yet. Another difference from *HDFS* is the use of *lineage* instead of a *replication* to provide fault tolerance.

In order to benchmark *Spark* performance Li et al. (2015) proposed a set of jobs. These jobs cover various APIs of *Spark* but lack data-processing application like the one we evaluation in the work. Li et al. (2015) also report that over-provisioning CPU capacity does not help, but we found that this is only true if job is CPU-bound.

A survey by Armbrust et al. (2015) reported improvements that were made to *Spark* performance over time. One area of performance issues is due to rich API, users are confused between performance of different transformations. A lot of improvements in memory management and networking layers were made: switching to *Netty* framework for high-performance networking, Zero-copy I/O, Off-heap network buffers, parallel data fetch. As a result *Spark* outperforms Hadoop on *Daytone GraySort* (Xin et al. 2014) by a factor if 2.5 using 10-x fewer nodes.

2.1.4 Choosing Hardware Resources

Jim Gray predicted in Gray & Graefe (1997) and Gray (2007) popularity of memory-based databases, so *Spark's RDD* (Zaharia et al. 2012) and *Tachyon* (Li et al. 2014) are examples of such systems. Hence the particular attention should be on the *DRAM* capacity when choosing resources.

Appuswamy et al. (2013) claimed that scaling up rather than out helps performance since high-end servers are getting cheaper. But authors observed that *Hadoop* does scale-up poorly. Authors noticed that few problems stems from the platform that *Hadoop* is built on *Java (JVM)*, for example the high overhead of starting up the new *JVM* and huge garbage collection overhead on large heaps. Appuswamy et al. (2013) proposed to use extra memory as a *RAM-disk* in order to speed up the shuffle operation. Authors also showed that *SSD* disks improve performance per $ by 60% for 16-node cluster. These observations reinforce points made by Gray (2007) that storage is moving into main memory.

2.2 Survey of Key Components of Cluster Management Systems

2.2.1 Distributed Scheduling

The key component to efficient resource utilization is a *Distributed Scheduler*. These schedulers are responsible for allocating particular tasks to nodes (usually a single server or VM). An early work by Zhou (1992) highlighted few benefits that are applicable to the cloud infrastructure today, such as supporting **single-system view** (i.e. few hosts are operated as one computer). This core principle is valid for most well known systems: *Apache Mesos* (Hindman et al. 2011) and *Google Omega* (Schwarzkopf et al. 2013). Another key principle is realization that clusters consists for heterogeneous systems, unlike *High Performance Computing (HPC)* clusters.

One more early work by Waldspurger & Weihl (1994) attempted to use probabilistic randomized scheduling for single operating system. Authors proposed to use the same scheduler for *I/O*, Network and CPU scheduling. Similar scheduler design is currently used in a few modern systems like *Quasar* (Delimitrou & Kozyrakis 2014).

Quasar improves scheduling by removing resource reservation from users. That way, users only specify requirements in terms of performance metrics, instead of shares of physical resources. Authors reported an improvement of 47% in utilization running on Public Cloud.

One of the first systems that was designed to run in a data-center - *Dryad* (Isard et al. 2007) made few important discoveries. Authors made an assumption, that system has high performance networking and is under one administrative domain, this is typical for modern data centers. Few aspects of scheduling can be relaxed, compared to LSF system Zhou (1992). Authors of Dryad system used centralized job scheduler, which a scalability bottleneck as recognized by Schwarzkopf et al. (2013).

Schwarzkopf et al. (2013) distinguished 3 kinds of schedulers: monolithic, tow-level and shared-state. Authors also claimed that shared-state scheduler is the most scalable and efficient. Yet Hindman et al. (2011) relied on two-level scheduler for its predictable behavior. Earlier approaches were based on network flow algorithms, such as Isard et al. (2009). The schedulers were not scalable enough for large clusters. Delimitrou et al. (2014) is an improved version of scheduler presented by Isard et al. (2009).

Ananthanarayanan et al. (2012) found that elasticity can increase utilization of infrastructure by splitting tasks into sub-tasks dynamically batch jobs can be assigned to

various types of hardware. Unfortunately it does not apply to service-like types of workloads.

2.2.2 Resource Allocation

In classic *Infrastructure as a Service (IaaS)* environments a unit of allocation is a *Virtual Machine (VM)*. Hindman et al. (2011) demonstrated that such allocation is too coarsely grained and lead to under-utilised capacity. Nguyen et al. (2013) proposed to use *wavelet* analysis and dynamic VM cloning to reduce the ratio of over provisioning capacity by 3.42 times.

Raman et al. (1999) claimed that problem with allocation algorithms is a centralized allocator, which does not have all information available on every node. Instead, they propose to use distributed resource allocation. Their system was designed to work atop of set of workstations, rather than in a data-center. What they also found, is that providing extensive language to end users may cause conditions under which none of the tasks can execute. This issue was already reported in Schwarzkopf et al. (2013), where user can specify constraints in terms of latency and time, rather than bytes and cpu-cycles.

Ghodsi et al. (2011) improved allocation in *Mesos* (Hindman et al. 2011) by introducing statistical strong metric, which combined not one type of resource (e.g. CPU) but all resources available for allocation. This approach is also an improvement over allocation used in *Dryad* (Isard et al. 2007). The disadvantage of presented metric that it does not take into account resource fragmentation, unlike Verma et al. (2014). Also, Verma et al. (2014) called out the fact that every allocation algorithm depends on underlying environment, because cluster, grids and data-centers have different requirements.

The key observation made by Mishra et al. (2010) is that there are two kinds of workloads in data-center: that most of resources are consumed by few long-running tasks. Another technique they used is to normalize resource usage for different types of resources, such as disk or cpu to the same numerical domain.

Another point of view on constraints is Sharma et al. (2011) work, where authors claimed that placement of tasks plays important role in performance, this point of view is shared by Mishra et al. (2010). Author composed a metric that represents how utilization is affected for every host in the cluster.

Zhang et al. (2011) work suggested to use simple metric to predict utilization of cluster, but this does not take into account that size of cloud infrastructure is itself elastic and

can be increased on demand. This makes it challenging to predict future utilization based on previous workload traces.

The need for dynamic resource allocation is clearly illustrated in *Apache Hadoop YARN* resource allocation manager (Vavilapalli et al. 2013). This approach is adopted by the majority of recent researches. Quasar improved this by introducing machine-learning into the allocation process (Delimitrou & Kozyrakis 2014).

2.2.3 Resource Isolation

Modern infrastructure relies on hardware and system *Virtualisation* to provide efficient mechanism to constraint resources. When co-locations happens inside one Virtual Machine (VM) it is essential to efficiently isolate different types of tasks: batch jobs and user-facing services.

CPU, memory, disk and network are essential resources and must be partitioned. An early study by Engler et al. (1995) proposed to delegate resource management and protection from operating system level to application layer. This way, each application can work with abstraction of a particular hardware component. This is similar to full virtualization approach for both at a fine-grained level. The problem with this approach is performance. Because of the performance, most of these functions should be performed in hardware. Only recently CPU vendors added basic virtualization support.

To deal with the imperfect isolation, Zhang et al. (2013) claimed that the key task of such isolation is to maintain low-latency of time sensitive customer facing tasks. Their metric *Cycles Per Instruction (CPI)* is used to describe latency of CPU. They also proposed to use throttling an artificial slowdown to resolve isolation problems. System proposed by Zhang et al. (2013) can also detect anomalies via collection of metrics and machine learning.

The key observation is that, when one program runs many times it is possible, using statistical methods, to obtain its *execution profile*. This execution profile can be used to detect anomalies in program's performance. Unfortunately this approach does not take into account that program code will evolve over time, and its profile will change with every new major version.

2.2.4 Data Locality

Data center networking has changed dramatically to support demand of cloud customers (Barroso et al. 2013). One of the core requirements is data-locality. As observed by Gray & Graefe (1997), the data that has to be delivered to the CPU creates the bottleneck for the computation. Hence this assumption has to be built into the scheduling algorithm of cluster management framework. *Apache Mesos* (Hindman et al. 2011) addresses this issue with delayed scheduling and reports 95% data-locality with 5 second delays in scheduling.

In contrast to this *Google Omega* (Schwarzkopf et al. 2013) does not report any optimizations related to data-locality. More recent systems like *Quasar* uses machine learning to train scheduler based on collected performance metrics from running jobs (Delimitrou & Kozyrakis 2014). This is clearly the most advanced approach currently in use.

Earlier systems like *Autopilot* (Isard 2007) do not focus on data locality at all, moreover they require an application running in *Autopilot* to be aware of it and replicate its state explicitly. Some systems like *Tarcil* (Delimitrou et al. 2014) ignore data locality completely, as a result tremendous improvement is seen in cluster utilization, because a significant proportion of time is spend on pulling data required for computation. Detecting this kind of "false" utilization is essential to identify inefficiencies.

2.2.5 Fault Tolerance

Cluster management system is a critical part of infrastructure, that must be available and should not have single point of failure (SPoF). Early systems like *Autopilot*(Isard 2007) indeed had a single point of failure. *Autopilot* rely on heart-beats that are aggregated on a *DeviceManager* which is a potential SPoF. Recent systems like *Mesos* (Hindman et al. 2011) and *Omega* (Schwarzkopf et al. 2013) employ distributed consensus algorithms, like *Paxos* to provide automatic fail-over. Quasar (Delimitrou & Kozyrakis 2014) in addition to using *Paxos* also uses master-slave replication as well. With automatic fail-over it is important to point that the state of the system is eventually consistent (Schwarzkopf et al. 2013).

Chapter 3

Design

This chapter describes our approach for migrating the application. Section 3.1 introduces the *JournalProcessor* application. In Section 3.2 we propose our methodology for migration. Section 3.3 presents the design of the experiment to evaluate correctness and performance for *Spark* version of *JournalProcessor*. Finally, in Section 3.4 we see how costs for consumed compute resources can be estimated.

3.1 Workload Description

The *JournalProcessor MapReduce* application is a batch log processing job that performs the following major steps (see subsection 3.1.1 for definitions):

1. Time-based aggregation
2. De-duplication
3. Index generation
4. Chunk generation

Figure 3.1 illustrates the flow of data in the *JournalProcessor*.

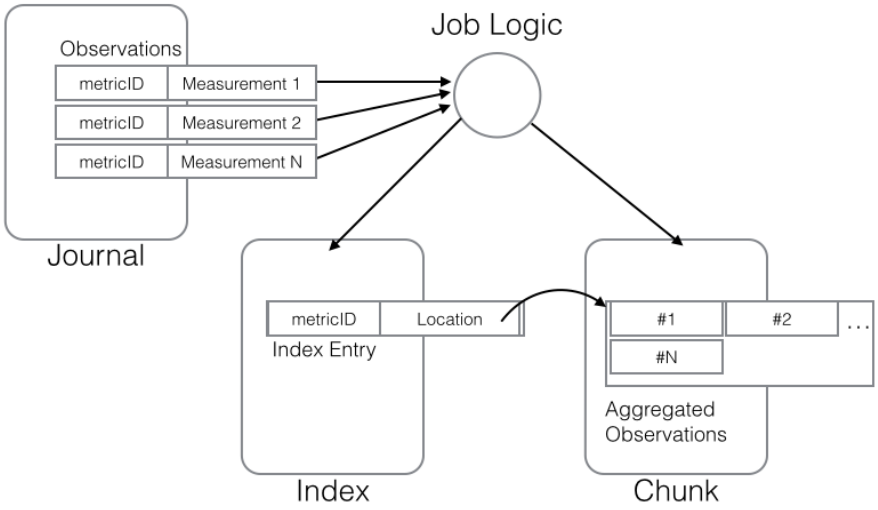

Figure 3.1: Overview of the JournalProcessor application

3.1.1 Terminology

To better describe the dataset we need to define the some terminology.

- *MetricID* – 16-bytes *SHA-1* hash-code of a published metric.
- *Timestamp* – 4-bytes *POSIX time* format, represents a moment in time.
- *Histogram* – Data structure that allows space-efficient representation of multiple *measurement* over time Datar et al. (2002), Kopp et al. (2013).
- *Measurement* – 8-bytes double in *IEEE-754* format or a *histogram*.
- *Observation* – represents a *measurement* of a *metricID* at a moment in time (*timestamp*).
- *Statistic* – a statistic derived from one or more *measurement*.
- *Aggregated Observation* – a combination of multiple *measurements* for the same *metricID* combined into a set of *statistics* or a *histogram*.
- *Journal* – file in an *Apache Avro* format with a sequence of observations.
- *Time Series* – a sequence of *measurements* ordered by their *timestamps* in an ascending order.
- *Chunk* – a binary file that contains a sequence of aggregated observations.

- *Index* – a binary *B-Tree* based mapping between metricID and sequence of aggregated observation that for this metricID.
- *Deduplication Token* – unique number in a *long* used to deduplicate observations, during aggregation.
- *Unit Type* – unit type of a single observation represented as a string (e.g. "Bytes").

3.1.2 Time-based Aggregation and Deduplication

Aggregation is performed based on the observation's timestamp. All aggregation occurs within a one-minute bucket with the timestamp of aggregated observation set to the start of minute of the first observation. Only observations with the same metricId and unit type are aggregated together. Every observation in a signle aggregation bucket must have unique deduplication token, othewise it gets discarded as a duplicate.

The aggregation process produces basic set of descriptive statistics. These statistics are: *sampleCount* – number of observations that were aggregated; *max* – maximum of all measurement of aggregated observations; *min* – minimum of all measurement of aggregated observations; *sum* – sum of all measurements of aggregated observations; *avg* – arithmetic average measurement of all aggregated observations.

The important mathematical property of above statistics is that they all can be performed by a *symmetric function*, which can be defined as: *A symmetric function of n variables is a function that value is unchanged by any permutation of its arguments.*

The requirement for a statistic to be computed by a *symmetric function* is strict and reduces the number of available statistics the algorithm can produce. One requirements that system has – is to support percentiles. This implemented via maintaining an *exponential histogram* Datar et al. (2002) for every aggregated observation.

3.1.3 Index and Chunk Generation

After aggregation and deduplication is performed this data is stored in the AWS S3. Output data split into two sets of objects in AWS S3: indexes and chunks. The output data is optimized to respond to the following query: *for a given metricID, unit type, and a period of time return all aggregated observations.* It order to be able to provide this data efficiently, size of a single index file is essential. How the size of index is picked is out of scope of this work.

3.1.4 Format of Input and Output Data

Input data represented as a sequence of journals in a single AWS S3 bucket. Journal is a binary file that contains records in the following format:

```
{"namespace": "job.metrics",
 "name": "AvroDatapoint",
 "type": "record",
"fields": [
  {"name": "metricId", "type": "string"},
  {"name": "aggregationId", "type": "int"},
  {"name": "unit", "type": "string"},
  {"name": "timestamp", "type": "long"},
  {"name": "count", "type": "double"},
  {"name": "min", "type": "double"},
  {"name": "max", "type": "double"},
  {"name": "sum", "type": "double"},
  {"name": "distribution", "type": [ "bytes", "null" ] }]
}
```

Listing 3.1: Avro Schema of a journal record

Each record is compressed using *GZIP*. Set of records represented as a *TAR* archive.

Output data represented as a set of chunk files in AWS S3. Each chunk in the sequence of records in the following format:

```
 {
    "timestamp": "long",
    "count": "double",
    "min": "double",
    "max": "double",
    "sum": "double",
    "unit": "string"
 }
```

Listing 3.2: Format of a chunk record

Along with chunks, job generates a set of index files with the following format:

```
{ metricID: [
    {
      "startTime": "timestamp",
      "endTime": "timestamp",
      "s3ObjectName": "string",
      "s3ObjectOffset": "long",
      "s3ObjectLength": "long"
    }
  ]
}
```

Listing 3.3: Format of a chunk record

The index stored as *B-Tree* data structure where key is metricID.

3.2 Migration Methodology

We propose the following methodology:

- Represent MapReduce job as a *Direct Acyclic Graph (DAG)* in such way that every transformation to the data is a vertex in the graph.
- For every transformation, record the input and output data-types.
- Replace the MapReduce transformations with one or more Spark transformation that matches the input/output data-types.

3.3 Experiment Design

The setup of an experiment is illustrated in Figure 3.2. We begin with constructing the input dataset by sampling the full dataset they MapReduce job process. To do that we use random sampling, to ensure that journals of all sizes are represented in the input. Then we execute MapReduce and Spark solutions using the same AWS EMR cluster configurations. After both job completed we validate the output data for its correctness. Then compare collected cluster utilization metrics metrics.

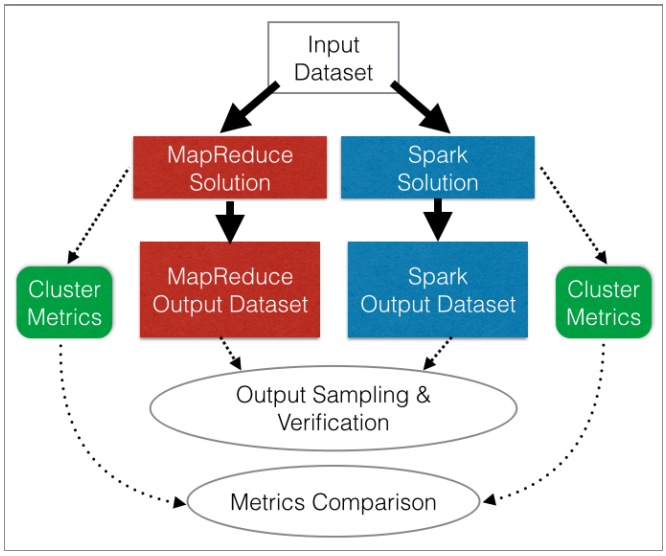

Figure 3.2: The steps of experiment

3.3.1 Compute and Storage Resources

To run both jobs we use 20 EC2 instances of *c3.8xlarge* type. Each instance has 32 cores, 60 GiB of RAM and 2 x 320 SSD drives. The cost per hour varies over time and AWS region, current price can be obtained at https://aws.amazon.com/ec2/instance-types/.

All computing resources utilize public AWS services. AWS EC2 On-Demand instances are managed by AWS EMR service.

3.3.2 Sampling the Input Dataset

In order for evaluation to be realistic, the sample data-set must be representative. We have picked a single run of an existing MapReduce job. From that run we collected the set of journals job has processed. Then, we sampled this data set to get 10% of all journals for evaluation.

As we can see from Figure 4.1 the majority of journals have size of 60, 135 or 210240 megabytes. Since for our experiment we want to pick 10% of journals, some random sampling technique must be applied.

One approach to get 10% of journals is to randomly pick N times number in the interval between 0 and 38,423 (i.e. total number of journals) and include journal with that index into sample. The problem with this approach is that given that sizes are distribute not evenly with three peaks around 60, 135 and 225 megabytes, most of the sampled journals will have these sizes.

In order to avoid this we partitioned list of journals into set of 31 buckets, with every bucket 15 megabytes in size. Then, we iterated over the set of buckets few times, randomly picking on every iteration journal from each bucket, until we have reached the 10% (see Listing A.3 for a code listing).

3.3.3 Validation of Output Dataset

In order to validate the correctness of the Spark solution we verify that set if metricIDs in the index files are the same for both solutions. Then we random sample 10% of metricIDs from the index. For that 10% of metrics we fetched its aggregated observations from the chunks and compare. Since input data is the same for both solutions we expect that this verification approach would yield equal results.

3.3.4 Metrics Collection

To compare usage of cluster resources we use feature of AWS EMR that allows to enable *Ganglia monitoring system* Massie et al. (2004) for the cluster. Ganglia provides aggregated metrics for total cluster CPU, Memory and Network utilization. For CPU

and Memory Ganglia also shows the maximum available capacity. We use these high-level metrics to compare cluster's resource utilization between MapReduce and Spark.

The granularity of metrics varies in Ganglia depending on the age of metrics. For last hour it prduces 1-minute metrics, after 1-hour all metrics available at 3-minute resolution.

Since both system run on top of *Hadoop YARN* Vavilapalli et al. (2013), we compare the following YARN metrics which are exposed via AWS CloudWatch and emitted by AWS EMR: containers allocated, containers reserved, containers pending.

3.4 Cost Estimation

There two main types of resources compute allocated via AWS EC2 instances and storage allocated via AWS S3. The cost on network transfer is not significant, since all data transfer happens inside AWS network. AWS does not charge for this type of traffic. Both solution consume the same amount of storage in S3. That means AWS EC2 – dominates the overall costs. AWS change compute resources of a per hour basis. Hence the cost of a single run can be calcualted using Equation 3.1.

$$Cost = PricePerHour \times NumberOfInstances \times \lceil JobRunimeInHours \rceil \tag{3.1}$$

To reduce the AWS S3 storage costs we use AWS S3 *Lifecycle Management Policies*. This allow us to delegate reclaim of storage to S3 itself.

AWS EMR terminates AWS EC2 instances it has provisioned when cluster finishes its work. AWS EC2 instances are charged by an hour boundary, this leads to some inefficiencies in compute resources in case cluster terminates not at the end of an hour.

The data does not leave the AWS network, hence there are no additional costs for data transfer.

Chapter 4

Implementation

An important part of this work is the implementation of the *Spark* version of the *JournalProcessor* job, described generally in the previous chapter. In this chapter we present details of the implementation. Section 4.1 provides details of the input sampling. In Section 4.2 we discuss *MapReduce*-based implementation of *JournalProcessor*. Finally, we looks into *Spark* version of the application in Section 4.3.

Before we begin, we must note that *MapReduce* version of *JournalProcessor* is a proprietary system, therefore we are not able to present its full source code. The full source code of the *Spark*-based application is available in Appendix C.

Versions of Software Used

For running MapReduce solution we used *Apache Hadoop 2.4* stack, the application itself was written using *Java7 SDK*. Spark solution uses *Apache Spark 1.3.1* and uses the same version of Hadoop to execute in a cluster mode on top of *YARN*. The application itself written using *Scala 2.10.5*.

4.1 Implementation of Input Sampling

In the MapReduce solution, single iteration processed 38,423 journals. Size of Journal varies from 46 bytes to 460 MiB. The total volume of journals is 4.253 TiB.

Figure 4.1 is the diagram with distribution of journal sizes grouped into 15 MiB buckets. Note the logarithmic scale of Y-axis.

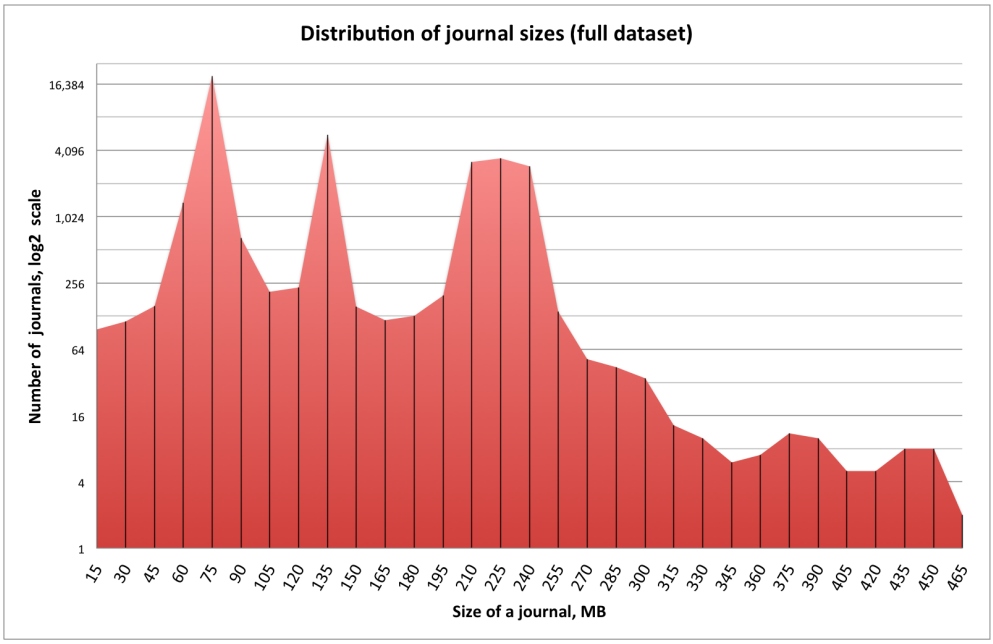

Figure 4.1: Distribution of a full input dataset (Log scale)

The resulted sample has the following distribution displayed in Figure 4.2. The sample contains 3,840 journals, with total size of 537.4 GiB. From Figure 4.2 we can conclude that sample includes journals of all sizes from the full dataset, hence the sample is representative.

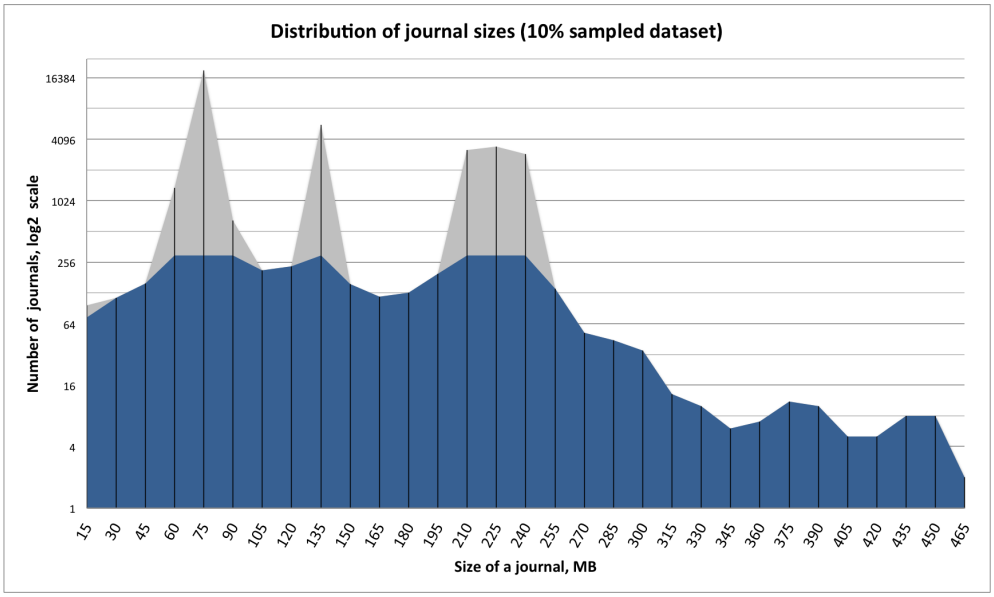

Figure 4.2: Distribution of a sampled input dataset overlaid on top of full dataset distribution (Log scale)

4.2 MapReduce Implementation

Listing Listing 4.1 represents the main components of a MapReduce job. The Hadoop MapReduce job is confugured by providing a set of hooks to the developer to include custom business logic.

The application's business logic concentrated in two Java classes: `Mapper` and `Reducer` which set via `job.setMapperClass()` and `job.setReducerClass()`.

The *secondary sort* pattern is implemented via three stages: `setPartitionerClass()`, `setSortComparatorClass()` and `setGroupingComparatorClass()`. The first stage is executed after mapper's output is written to disk, second stage executed by MapReduce internal shuffle process and the last stage is executed by reducer during construction of an input iterator.

```
Job job = new Job(conf, getJobName());
job.setJarByClass(...);

job.setInputFormatClass(...);

job.setMapperClass(...); // parsing input records

job.setMapOutputKeyClass(...);
job.setMapOutputValueClass(...);

job.setReduceClass(...); // business logic

job.setPartitionerClass(...); // secondary sort, mapper side
job.setSortComparatorClass(...); // secondary sort, shuffle side
job.setGroupingComparatorClass(...); // secondary sort, reducer side

job.setOutputKeyClass(...);
job.setOutputValueClass(...);
job.setOutputFormatClass(...);

result = job.waitForCompletion(true);
```

Listing 4.1: Components of a MapReduce job written in Java

4.2.1 Tuninig MapReduce Solution

The first issues that began clear during the implementation is a lack of sufficient parallelism. Only small portion of the cluster was utilized during the map phase of the job. The root cause of this issue was a HDFS data-locality property, YARN only scheduled map tasks on nodes where part of input split was present, but since the size of input splits was small - just a name of the journal to fetch the whole input was in a few HDFS blocks. In order to correct that, differemt splitting algorithm was used: `NLineInputFormat`. Also, the replication factor was set to maximum value. That gave a desired parallelism during the map phase.

Another problem was the deadlock of a Hadoop cluster. That was triggered by *reducers slow start* feature which starts reducers while mappers are still running. The deadlock condition happened when all containers allocated by YARN were running reducers and all these reducers were waiting an output of s single map task, which failed but could not re-run since there were no available containers. The solution was to increase `mapreduce.job.reduce.slowstart.completedmaps` to 1.0.

4.3 Spark Implementation

Apache Spark application, presented in listing Listing 4.2 written in Scala and has different structure than a MapReduce job. Instead of plugging custom classes, functional programming style is used to construct a dataflow which consists of a set of *Transformations* and ends with an *Action*. The size of code for Spark solution is an order of magnitude less that a MapReduce Java solution.

```
val sc = new SparkContext(conf)

val result = sc.parallelize(journalS3Locations)
  .flatMap(downloadAndParseJournals)
  .groupSort(Ordering) // secondary sort, map side
  .mapStreamByKey(performTemporalAggregation) // secondary sort, reduce side
  .mapPartitionsWithIndex(storeIndexAndChunksToS3)
  .collect
```

Listing 4.2: Components of a Spark job written in Scala

Listing 4.3 shows the structure of RDD, generated for a sequence of transformations in Listing 4.2. `ParallelCollectionRDD[0]` is the first RDD generated by splitting list of journal names from S3 into one file per partition to maximize the amount of `MapPartitionsRDD[2]` tasks which pull journals from *AWS S3*. Next there is a `ShuffledRDD[3]` which performs grouping observations per *metricID* and sorting observations by *timestamp*. We set number of partitions produced by shuffle to be *2432* in `MapPartitionsRDD[7]`. This was done to optimize the amount of memory used by tasks executing *mapPartitionsWithIndex* transformation.

```
  (2432) MapPartitionsRDD[7] at mapPartitionsWithIndex at App.scala:143 []
  |    MapPartitionsRDD[6] at mapPartitions at GroupSorted.scala:20 []
  |    anon$1[5] at RDD at GroupSorted.scala:80 []
  |    MapPartitionsRDD[4] at mapPartitions at PairRDDFunctions.scala:31 []
  |    ShuffledRDD[3] at ShuffledRDD at PairRDDFunctions.scala:29 []
  +-(3840) MapPartitionsRDD[2] at map at PairRDDFunctions.scala:29 []
      |    MapPartitionsRDD[1] at flatMap at App.scala:114 []
      |    ParallelCollectionRDD[0] at parallelize at App.scala:114 []
```

Listing 4.3: Structured of RDD generated by Spark

4.3.1 Secondary Sort

MapReduce solutions uses *secondary sory* feature of Hadoop to guarantee that reducers receive observation sorted by their timestamp. Unlike MapReduce, Spark has limited support for secondary sort, its implementation is still in progress as of Thursday 17th September, 2015: https://issues.apache.org/jira/browse/SPARK-3655. We implement the secondary sort, we used thrid-party library: https://github.com/tresata/spark-sorted.

Chapter 5

Evaluation

In this chapter we validate the *Spark*-based version of *JournalProcessor* application. Section 5.1 reports our observations of cluster utilization metrics collected by *Ganglia*, *CloudWatch*, *Spark* and *AWS EMR*. In Section 5.2 we present our analysis of observed metrics. Finally, we evaluate our experience developing application with *MapReduce* and *Spark* APIs in Section 5.3.

The reader may refer to Appendix B for additional data on cluster performance.

5.1 Observations

The execution time of the Spark job was *2 hours, 12 minutes* and execution time of the MapReduce job was *1 hour, 24 minutes*. **Spark job took 63,63% more time than a MapReduce**.

5.1.1 Ganglia Metrics

In Table 5.1 overall cluster utilization metrics are summarized. The normalized CPU Utilization is computed using the following formula: *(1 min load) / (number of CPUs)*. Figure 5.1 illustrates change of metrics over time.

	MapReduce		Spark	
statistic	Avg	Max	Avg	Max
1-min load_avg	215	471	880	1010
cpu utilization, %	28.1%	65.4%	72.0%	93.0%
memory utilization, GiB	282.6	1024	849.1	1126.4
network, bytes in, GiB	0.58	5.20	0.24	0.83
network, bytes out, GiB	0.50	5.30	0.19	0.97

Table 5.1: Summary of Ganglia cluster utilization metrics

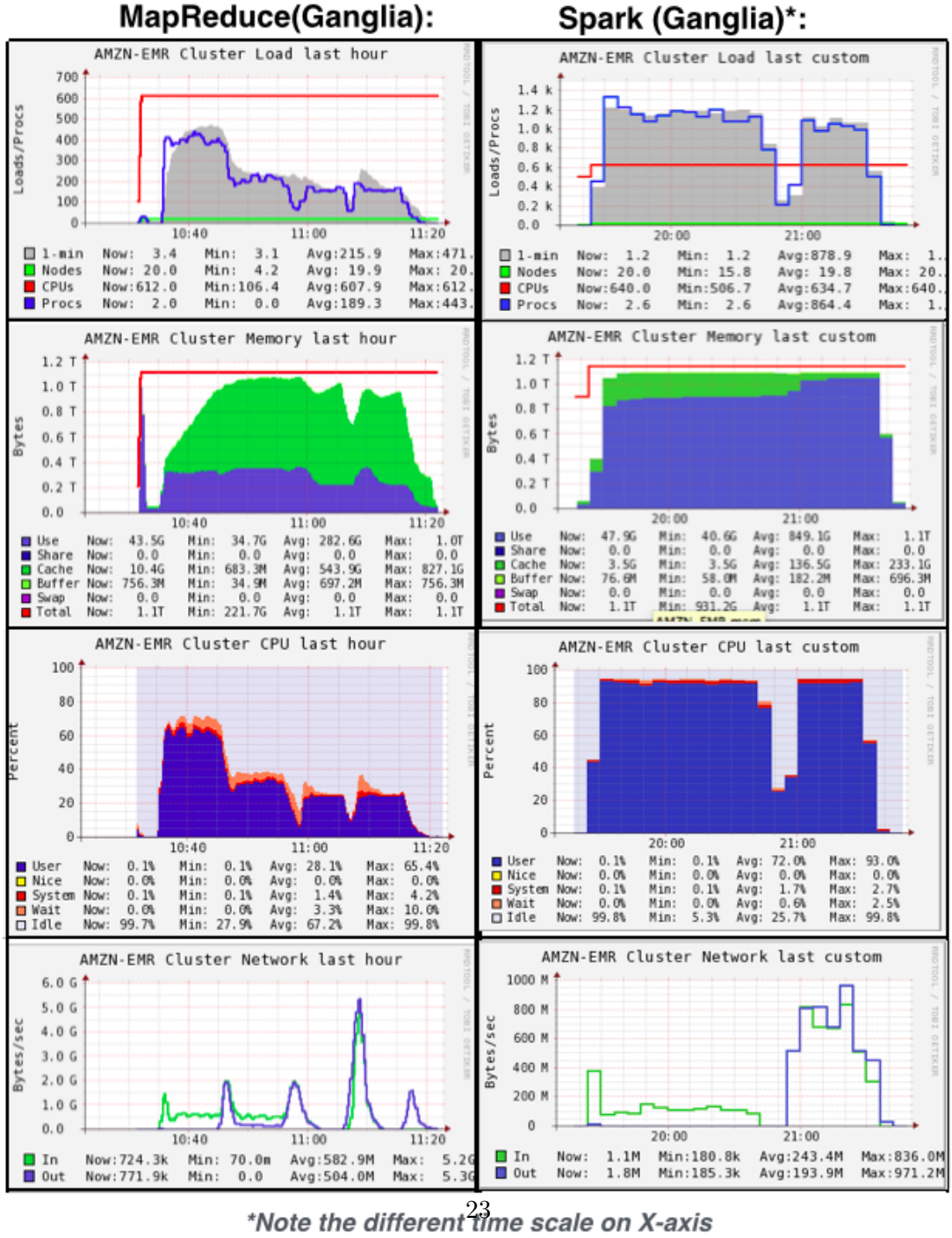

Figure 5.1: Overview of Ganglia metrics for MapReduce and Spark

5.1.2 CloudWatch Metrics

AWS EC2 collects basic metrics with 5-minute granularity for every running EC2 instance. Table 5.2 compares these metrics. The most notable difference can be seen in the CPU Utilization as shown on Figure 5.2. Rest of the CloudWatch metrics can be found in Appendix B.1. Unfortunately EC2 does not record metrics on memory utilization, but these metrics available in the previous section via Ganglia.

	MapReduce	Spark
statistic	Max	Max
CPU Utilization, %	88%	100%
Disk Read, Bytes	8,8 GiB	10 GiB
Disk Read Ops, count	135,000	240,000
Disk Write, Bytes	18 GiB	8,1 GiB
Disk Write Ops, count	280,000	126,000
Network In, Bytes	23.2 GiB	6.2 GiB
Network Out, Bytes	17 GiB	6.5 GiB

Table 5.2: Summary of CloudWatch metrics

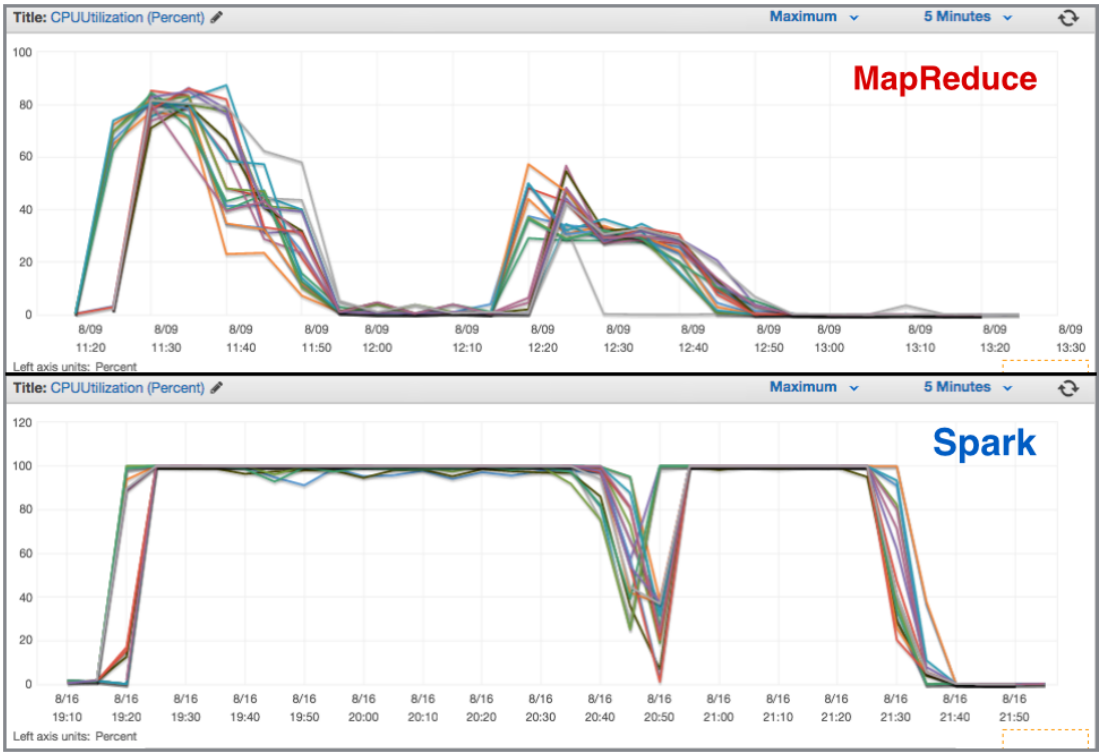

Figure 5.2: EC2 Instance CPUUtilization,%

5.1.3 I/O Performance

AWS EMR executes every 15 minutes various system monitoring commands like *top*, *vmstat*, *netstat* and *iostat*. Listing 5.1 shows the output of *iostat* command during the reducing part of the job.

```
...
avg-cpu: %user  %nice %system %iowait %steal %idle
         92.33   0.00   4.91   1.75   0.00   1.00

Device:        rrqm/s  wrqm/s    r/s    w/s  rsec/s  wsec/s avgrq-sz avgqu-sz await svctm %util
xvdap1           0.00    0.00   0.00   0.00    0.00    0.00    0.00    0.00   0.00  0.00  0.00
xvdb             0.00    0.00   0.00 2853.00    0.00 728184.00 255.23  136.25  46.36  0.35 100.00
xvdc             0.00    0.00   0.00 1342.00    0.00 342976.00 255.57  137.32 102.85  0.75 100.00

avg-cpu: %user  %nice %system %iowait %steal %idle
         88.59   0.00   3.46   2.89   0.03   5.03

Device:        rrqm/s  wrqm/s    r/s    w/s  rsec/s  wsec/s avgrq-sz avgqu-sz await svctm %util
xvdap1           0.00    0.00   0.00   0.00    0.00    0.00    0.00    0.00   0.00  0.00  0.00
xvdb             0.00    0.00   0.00 1544.00    0.00 394544.00 255.53  144.37  95.42  0.65 100.00
xvdc             0.00    0.00   0.00 1384.00    0.00 353888.00 255.70  151.89 109.57  0.72 100.00
...
```

Listing 5.1: MapReduce Disk I/O Utilization during the reduce phase (iostat tool)

```
...
avg-cpu: %user  %nice %system %iowait %steal %idle
         52.08   0.00   7.89   2.57   0.00  37.46

Device:        rrqm/s  wrqm/s    r/s    w/s  rsec/s  wsec/s avgrq-sz avgqu-sz await svctm %util
xvdap1           0.00    0.00   0.00   0.00    0.00    0.00    0.00    0.00   0.00  0.00  0.00
xvdb             0.00    0.00   0.00   1.00    0.00    8.00    8.00    0.00   0.00  0.00  0.00
xvdc             0.00    1.00   0.00 420.00    0.00 106768.00 254.21  141.74 361.56  2.38 100.00

avg-cpu: %user  %nice %system %iowait %steal %idle
         52.96   0.00   4.25   1.44   0.00  41.35

Device:        rrqm/s  wrqm/s    r/s    w/s  rsec/s  wsec/s avgrq-sz avgqu-sz await svctm %util
xvdap1           0.00    0.00   0.00   9.00    0.00  168.00   18.67    0.00   0.00  0.00  0.00
xvdb             0.00    0.00   0.00   0.00    0.00    0.00    0.00    0.00   0.00  0.00  0.00
xvdc             0.00    1.00   0.00 281.00    0.00 71664.00 255.03  128.65 434.06  3.56 100.00
...
```

Listing 5.2: Spark Disk I/O Utilization during the shuffle (iostat tool)

5.1.4 Spark Executor Metrics

Additionally we were able to collect details metrics for Spark executors: Figure B.7 and Figure B.8. We also provide a snapshot of CPU Utilization, during the execution of a *Stage 0* (i.e. the map phase) of a Spark solution in Figure B.9.

5.2 Analysis

As we can see from this graph Spark was able to utilize all provisioned CPU and memory.

It is clearly visible on the Figure 5.1 two stages on a job: before shuffle and after. In both MapReduce and Spark CPU was better utilized during the mapping phase, when journals are fetched and parsed from AWS S3. Yet, both jobs left over half of the CPU resources not used.

Looking at memory utilization, Spark was able to use more memory than MapReduce and memory utilization pattern is different compared to MapReduce. We believe that this is caused by the approach Spark request containers from YARN, i.e. single container per node, which persists during the execution of a whole job.

5.2.1 MapReduce I/O Performance

To improve utilization of CPU and memory we reduced memory allocated by YARN to mapper and reducers, by changing `mapreduce.map.memory.mb` from 3 GiB to 2 GiB and lowering `mapreduce.reduce.memory.mb` from 6.6 GiB ti 4.6 GiB. As a result we got close to 100Memory utilization went up by only 10But during the reduce phase CPU utilization went down to 20

The root cause of this was the bottleneck of disk I/O on the reducers. Listing 5.1 is a snippet from *instance-state* log file that AWS EMR collects every 15 minutes for every node of a cluster. As we observed from Listing 5.1 SSD disk on the instances experience a significant write load of over 3000 IOPS.

Tan, Fong & Liu (2014) showed that using SSD disks significantly improves performance, moreover we share the load between two distinct SSD drives `xvdb` and `xvdc`. *Even with these improvements SSD disks do not provide enough capacity to serve data fast enough to the CPU for processing.*

5.2.2 Spark I/O Performance

Similar disk I/O bottleneck was observed in Spark as well. Except as seen from Listing 5.2 only one disk was utilized instead of two. We belive that this is due to misconfiguration of OS. Our observations contradict claim that Spark is an in-memory system, since we found that as in MapReduce disk I/O throughput is a bottleneck.

5.2.3 On Non Uniform Distribution of Load

Since journal represented as a *TAR* archive, of one or more *ZIP* files with custom headers between which prefix each file in a *TAR* archive. It is not possible for Hadoop I/O to split the file into equal set of blocks, i.e. mapper cannot process just a portion of a journal file, only the whole file. Given that distribution of journals varies from few magabytes up to 460 MiB, load on some nodes that fetch journals from S3 and parse them (mappers in case of MapReduce), is significantly higher. This limits the parallelism of the cluster. Since all data for single *metricID* must be processed by exactly one reducer or in case of Spark by the node that run first transformation in every stage.

5.2.4 Resource Allocation Using YARN

Both Spark and MapReduce run on top of Hadoop YARN resource manager Vavilapalli et al. (2013). YARN can manage two types of resources: *vCores* total number of CPU-cores in the cluster, and *memory* total volume of RAM in the cluster. YARN only enforces memory usage, i.e. the container will be terminated by a *NodeManager* if it attempts allocate more memory than allowed.

This complicates the configuration process of Spark running on YARN, since memory limits must be set separately in Spark configuration via `spark.executor.memory` and in YARN's configuration via `yarn.nodemanager.resource.memory-mb`. Another type constraint YARN enforces is *locality*, but in the *AWS EC2* there is no notion of locality, for example there is no way to guarantee that two EC2 instances will be provisioned in the same physical rack. The only placement constrain that exists is that all EC2 instances of the *AWS EMR* cluster must be in the same *Availability Zone* and within the same *Virtual Subnet*.

We found it difficult to compare YARN metrics between MapReduce and Spark since allocation algorithms are different for MapReduce and Spark. MapReduce allocates resources in two phases: Containers for map tasks requested upfront, but containers

for reduce tasks are not started until some portion of map tasks are finished. On the other hand, Spark request all containers for executors upfront. In our experiment we configured Spark to request one container per node. This allows Spark to run multiple tasks in the single container and JVM, thus reducing the startup overhead.

One more disadvantage of Spark we observed is that support only homogeneous clusters. Every executor get's the same amount of cores and memory. For out workload this reduced efficiency since tasks that fetch journals require less memory that tasks that aggregate observations.

5.3 MapReduce and Spark API

Both Apache Hadoop MapReduce and Apache Spark engines built on top of JVM virtual machine. The main different is that MapReduce developed in JVM native language Java, but Spark developed using Scala language, which is a functional programming language unlike object oriented Java. This makes APIs for both systems very different. In MapReduce developer provide classes to job configuration. In Spark developer uses higher-order functions (i.e. functions that accept another functions as agruments) and lazy computation.

In our solutions the size of code for Spark application is an smaller than the size of MapReduce solution by a factor of 10.

Chapter 6

Conclusion

In this thesis we have shown that migrating existing application from *MapReduce* to *Spark* is indeed possible, but doing so did not improved performance for the *Journal-Processor* application. Despite that we were able to configure *Spark* in such a way that it provided better cluster utilization compared to *MapReduce* solution.

We have found that *Apache Spark* is a work-in-progress system under heavy development. Moreover, we found that *I/O* subsystem was misconfigured in *Spark*. Due to that error the disk throughput was only 50% of *MapReduce*.

Two essential features that we were missing during the implementation are:

- The ability to efficiently read binary data from *Amazon S3*: `https://issues.apache.org/jira/browse/SPARK-6527`. This issue is open at the moment of writing.
- The lack of efficient *Secondary Sort* implementation: `https://issues.apache.org/jira/browse/SPARK-3655`. This issue is in progress at the moment of writing.

Implementing work-around solutions with sub-optimal performance resulted in significant increase of *Spark* application running time.

6.0.1 Monitoring of the Cluster

We have found that *Spark* provides great web-based user interface (UI) for debugging and monitoring cluster (for example Figure B.7 and Figure B.8). The important advantage over MapReduce is that Spark UI is available after job completes for post-analysis,

unlike *MapReduce* where *ApplicationMaster* UI is shut down as soon as a job completes.

During our experiment we found it difficult to collect metrics related to cluster utilization. We evaluated the following sources:

- Ganglia Monitoring system – detailed metrics with 1-minute granularity but only for the last hour. All metrics after 1 hour have 5-minute granularity. Ganglia scales poorly with the size of cluster. For example with the cluster size of 200 nodes, it takes few minutes to display aggregated metrics for the cluster.
- EMR Instance stage log – detailed log over node resource usage by executing OS commands like `top` or `iostat`. Commands are executed with a 15-minute interval which makes it hard to use this data for troubleshooting.
- EMR CloudWatch metrics – basic aggregated metrics for cluster utilization with 5-minute granularity.
- EC2 instance metrics – basic instance usage metrics, but lack memory utilization.

6.0.2 Debugging Application

In our experience developing and tuning Spark application was much easier than Hadoop Map-Reduce application. In *Spark* we were able to apply one transformation at a time and to test every function. In *MapReduce* we switched between map and reduce code few times to debug errors, when input data format changed during the implementation. We believe that *Spark* application has better architecture and modularity of the codebase.

6.0.3 Migration Best Practices

During the implementation we established the following best practices for migration batch log-processing application, that we believe are general and applicable to a broad range of MapReduce workloads.

Decompose MapReduce application into a Graph

We have found that it is useful to decompose the *MapReduce* application into a Directed Acyclic Graph (DAG). This allows to evaluate the migration plan in details before developing *Spark* application.

Identifying Shuffle Operation

Since shuffle transformation is the most expensive operation in terms of CPU and I/O load it is important to identify it early. One should consider migrating application in such a way that shuffle operation can be eliminated.

Choosing the Right Instance

We found that EC2 instance type we were using (c3.8xlarge) has poor balance between CPU processing and I/O throughput. We believe that identifying the expected CPU, memory, network and I/O load is essential *before* starting the migration work.

6.0.4 Observed Patterns

We identified the following *generic* patterns in both systems.

- Shuffle – During this transformation cluster is divided into two partitions, every node in one partition communicates with every node in another partition.
- Secondary Sort – Allows to impose an ordering over the sequence of records that is processed in every transformation (for example ordering records by *timestamp* field).
- Splittable Input – In order for every task to have input of the equal size (to maximize performance), the input data must be dividable into chunks of equal size. We believe that this will result in uniform load of the cluster and overall better utilization.

6.1 Future Work

We believe that Spark engine will mature and the functionality will stabilize over time. We see few areas where our research can be extended.

6.1.1 Fix Spark disk configuration

In our experiments Spark was using only one disk out of two. We believe this is an error, the disks must be grouped into *RAID0* or *RAID10* to maximize throughput.

6.1.2 Evaluate Different Instance Types

In our evaluation we look only on single type of EC2 instance. It it interesting to evaluate various options, particularly with high provisioned I/O setting in AWS EC2.

6.1.3 Execution Unmodified MapReduce on Spark

We believe that it is possible to introduce a *middleware* that is capable of executing unmodified MapReduce job with Spark engine by intercepting API calls from MapReduce and converting them into Spark API.

Bibliography

Ananthanarayanan, G., Douglas, C., Ramakrishnan, R., Rao, S. & Stoica, I. (2012), True elasticity in multi-tenant data-intensive compute clusters, *in* 'Proceedings of the Third ACM Symposium on Cloud Computing', SoCC '12, ACM, New York, NY, USA, pp. 24:1–24:7.
URL: *http://doi.acm.org/10.1145/2391229.2391253*

Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O. & Rowstron, A. (2013), Scale-up vs scale-out for hadoop: Time to rethink?, *in* 'Proceedings of the 4th annual Symposium on Cloud Computing', ACM, p. 20.

Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R. & Zaharia, M. (2015), 'Scaling spark in the real world: Performance and usability', *Proceedings of the VLDB Endowment* **8**(12).

Barroso, L. A., Clidaras, J. & Hölzle, U. (2013), 'The datacenter as a computer: an introduction to the design of warehouse-scale machines', *Synthesis Lectures on Computer Architecture* **8**(3), 1–154.

Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J. & Tian, Y. (2010), A comparison of join algorithms for log processing in mapreduce, *in* 'Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data', SIGMOD '10, ACM, New York, NY, USA, pp. 975–986.
URL: *http://doi.acm.org/10.1145/1807167.1807273*

Borthakur, D. (2007), 'The hadoop distributed file system: Architecture and design', *Hadoop Project Website* **11**(2007), 21.

Datar, M., Gionis, A., Indyk, P. & Motwani, R. (2002), 'Maintaining stream statistics over sliding windows', *SIAM Journal on Computing* **31**(6), 1794–1813.

Davidson, A. & Or, A. (2013), 'Optimizing shuffle performance in spark', *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* .

Dean, J. & Ghemawat, S. (2004), Mapreduce: Simplified data processing on large clusters, *in* 'Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6', OSDI'04, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL: *http://dl.acm.org/citation.cfm?id=1251254.1251264*

Delimitrou, C. & Kozyrakis, C. (2014), Quasar: resource-efficient and qos-aware cluster management, *in* 'Proceedings of the 19th international conference on Architectural support for programming languages and operating systems', ACM, pp. 127–144.

Delimitrou, C., Sanchez, D. & Kozyrakis, C. (2014), 'Tarcil: Reconciling scheduling speed and quality

in large, shared clusters'.
URL: *https://web.stanford.edu/group/mast/cgi-bin/drupal/system/files/2014.techreport.tarcil.pdf*

Engler, D. R., Kaashoek, M. F. & O'Toole, Jr., J. (1995), Exokernel: An operating system architecture for application-level resource management, *in* 'Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles', SOSP '95, ACM, New York, NY, USA, pp. 251–266.
URL: *http://doi.acm.org/10.1145/224056.224076*

Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S. & Stoica, I. (2011), Dominant resource fairness: Fair allocation of multiple resource types, *in* 'Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation', NSDI'11, USENIX Association, Berkeley, CA, USA, pp. 323–336.
URL: *http://dl.acm.org/citation.cfm?id=1972457.1972490*

Gray, J. (2007), 'Tape is dead, disk is tape, flash is disk, ram locality is king', *Gong Show Presentation at CIDR* .

Gray, J. & Graefe, G. (1997), 'The five-minute rule ten years later, and other computer storage rules of thumb', *SIGMOD Rec.* **26**(4), 63–68.
URL: *http://doi.acm.org/10.1145/271074.271094*

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S. & Stoica, I. (2011), Mesos: A platform for fine-grained resource sharing in the data center, *in* 'Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation', NSDI'11, USENIX Association, Berkeley, CA, USA, pp. 295–308.
URL: *http://dl.acm.org/citation.cfm?id=1972457.1972488*

Isard, M. (2007), 'Autopilot: Automatic data center management', *Operating Systems Review* **41**(2), 60–67.
URL: *http://research.microsoft.com/apps/pubs/default.aspx?id=64604*

Isard, M., Budiu, M., Yu, Y., Birrell, A. & Fetterly, D. (2007), Dryad: Distributed data-parallel programs from sequential building blocks, *in* 'Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007', EuroSys '07, ACM, New York, NY, USA, pp. 59–72.
URL: *http://doi.acm.org/10.1145/1272996.1273005*

Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K. & Goldberg, A. (2009), Quincy: Fair scheduling for distributed computing clusters, *in* 'Proceedings of 22nd ACM Symposium on Operating Systems Principles', Association for Computing Machinery, Inc., pp. 261–276.
URL: *http://research.microsoft.com/apps/pubs/default.aspx?id=81516*

Kopp, C., Mock, M., Papapetrou, O. & May, M. (2013), Large-scale online mobility monitoring with exponential histograms., *in* 'BD3@ VLDB', Citeseer, pp. 61–66.

Laney, D. (2001), '3d data management: Controlling data volume, velocity and variety', *META Group Research Note* **6**, 70.

Li, H., Ghodsi, A., Zaharia, M., Shenker, S. & Stoica, I. (2014), Tachyon: Reliable, memory speed storage for cluster computing frameworks, *in* 'Proceedings of the ACM Symposium on Cloud Computing', SOCC '14, ACM, New York, NY, USA, pp. 6:1–6:15.
URL: *http://doi.acm.org/10.1145/2670979.2670985*

Li, M., Tan, J., Wang, Y., Zhang, L. & Salapura, V. (2015), Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark, *in* 'Proceedings of the 12th ACM International Conference on Computing Frontiers', CF '15, ACM, New York, NY, USA, pp. 53:1–53:8.
URL: *http://doi.acm.org/10.1145/2742854.2747283*

Logothetis, D., Trezzo, C., Webb, K. C. & Yocum, K. (2011), In-situ mapreduce for log processing, *in* '2011 USENIX Annual Technical Conference (USENIX ATC11)', p. 115.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. & Czajkowski, G. (2010), Pregel: a system for large-scale graph processing, *in* 'Proceedings of the 2010 ACM SIGMOD International Conference on Management of data', ACM, pp. 135–146.

Massie, M. L., Chun, B. N. & Culler, D. E. (2004), 'The ganglia distributed monitoring system: design, implementation, and experience', *Parallel Computing* **30**(7), 817–840.

Mishra, A. K., Hellerstein, J. L., Cirne, W. & Das, C. R. (2010), 'Towards characterizing cloud backend workloads: insights from google compute clusters', *ACM SIGMETRICS Performance Evaluation Review* **37**(4), 34–41.

Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P. & Abadi, M. (2013), Naiad: A timely dataflow system, *in* 'Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles', SOSP '13, ACM, New York, NY, USA, pp. 439–455.
URL: *http://doi.acm.org/10.1145/2517349.2522738*

Nguyen, H., Shen, Z., Gu, X., Subbiah, S. & Wilkes, J. (2013), Agile: Elastic distributed resource scaling for infrastructure-as-a-service, *in* 'Proc. of the USENIX International Conference on Automated Computing (ICAC13). San Jose, CA'.

Raman, R., Livny, M. & Solomon, M. (1999), 'Matchmaking: An extensible framework for distributed resource management', *Cluster Computing* **2**(2), 129–138.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. & Wilkes, J. (2013), Omega: flexible, scalable schedulers for large compute clusters, *in* 'Proceedings of the 8th ACM European Conference on Computer Systems', ACM, pp. 351–364.

Sharma, B., Chudnovsky, V., Hellerstein, J. L., Rifaat, R. & Das, C. R. (2011), Modeling and synthesizing task placement constraints in google compute clusters, *in* 'Proceedings of the 2nd ACM Symposium on Cloud Computing', ACM, p. 3.

Tan, W., Fong, L. & Liu, Y. (2014), Effectiveness assessment of solid-state drive used in big data services, *in* 'Web Services (ICWS), 2014 IEEE International Conference on', pp. 393–400.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B. & Baldeschwieler, E. (2013), Apache hadoop yarn: Yet another resource negotiator, *in* 'Proceedings of the 4th Annual Symposium on Cloud Computing', SOCC '13, ACM, New York, NY, USA, pp. 5:1–5:16.
URL: *http://doi.acm.org/10.1145/2523616.2523633*

Verma, A., Korupolu, M. & Wilkes, J. (2014), Evaluating job packing in warehouse-scale computing, *in* 'IEEE Cluster', Madrid, Spain.

Waldspurger, C. A. & Weihl, W. E. (1994), Lottery scheduling: Flexible proportional-share resource

management, *in* 'Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation', OSDI '94, USENIX Association, Berkeley, CA, USA.
URL: *http://dl.acm.org/citation.cfm?id=1267638.1267639*

Xin, R., Deyhim, P., Ghodsi, A., Meng, X. & Zaharia, M. (2014), 'Graysort on apache spark by databricks'.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. & Stoica, I. (2012), Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *in* 'Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation', USENIX Association, pp. 2–2.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010), Spark: cluster computing with working sets, *in* 'Proceedings of the 2nd USENIX conference on Hot topics in cloud computing', pp. 10–10.

Zhang, Q., Hellerstein, J. L. & Boutaba, R. (2011), Characterizing task usage shapes in googles compute clusters, *in* 'Large Scale Distributed Systems and Middleware Workshop (LADIS11)'.

Zhang, X., Tune, E., Hagmann, R., Jnagal, R., Gokhale, V. & Wilkes, J. (2013), Cpi 2: Cpu performance isolation for shared compute clusters, *in* 'Proceedings of the 8th ACM European Conference on Computer Systems', ACM, pp. 379–391.

Zhou, S. (1992), Lsf: Load sharing in large heterogeneous distributed systems, *in* 'I Workshop on Cluster Computing'.

Appendix A

A.1 Script to Sample the Input Dataset

Input dataset consists of set of *Journals* located in an *Amazon S3* bucket. Along with journals there are *Manifest* files that contain list of journals processed by a single batch job. A.1 shows a snipped of a single Manifest file.

```
[
  {
     "bucket": "journals",
     "length": 256645126,
     "key": "zzZ1y_v06_20150623T024600Z_7edd3b29-562f-4258-af9b-80c109fef60e.input"
  },
  ...
  {
    "bucket": "journals",
    "length": 77076486,
    "key": "zzp7D_v06_20150622T210855Z_2371bafb-bbd2-4874-9a28-6fa3a68bd3d1.input"
  }
]
```

Listing A.1: Sample of a Manifest file

Script in A.2 copies *Amazon S3* objects between buckets.

```
#!/usr/bin/env bash

set -e

n=10
while read size object; do
  ( aws s3 cp --region us-east-1 \
    s3://journals-us-east-1/$object \
    s3://lvsl-thesis-journals/ ) &

  n=$(($n - 1))
  if [ $n -eq 0 ]; then
    wait
    n=10
  fi
done < ./sampled_journals.txt

wait
echo "Done!"
```

Listing A.2: Script to copy s3 objectsd between bucket in bulk

Script in A.3 used to pick the random subset of journals from a manifest file:

```
import sys
import json
import random
import bisect

from collections import namedtuple
from itertools import cycle

Journal = namedtuple('Journal', 'size s3_object')

def main(percent):
    journals = []
    for line in sys.stdin:
        entry = json.loads(line)
        # skip injected data
        if 'injected' in entry['key']:
            continue
        journals.append(Journal(int(entry['length']), entry['key']))

    number_of_journals = len(journals)
    sample_size = number_of_journals * 0.1 # 10%

    max_size = max(journals).size
    number_of_buckets = 31
    size_of_bucket = 15 * 1048576 # 15MiB

    buckets = [size_of_bucket * i for i in range(1, number_of_buckets + 1)]
    bins = [[] for _ in range(number_of_buckets)]

    for j in journals:
        index = bisect.bisect_left(buckets, j.size)
        bins[index].append(j)

    # shuffle each bin
    for journals_in_bin in bins:
        random.shuffle(journals_in_bin)

    # get random journals
    bin_indexes = cycle(range(number_of_buckets))
    sampled_journals = []
    while len(sampled_journals) < sample_size:
        while True:
            index = next(bin_indexes)
            if not bins[index]: # bin already empty
                continue
            sampled_journals.append(bins[index].pop())
            break

    print '\n'.join("%s %s" % (j.size, j.s3_object) for j in sorted(sampled_journals ↩
        ))

if __name__ == '__main__':
    main(sys.argv[1])
```

Listing A.3: Python script for sampling input data

A.2 Script to Run Spark on Amazon Elastic Map Reduce (EMR)

```
#!/usr/bin/env bash
set -e
if [ -n "$DEBUG" ]; then
  set -x
fi

if [ ! -f "$1" ]; then
  echo "Uber JAR does not exists: '$1'" >&2
  exit 201
fi
if [ -z "$2" ]; then
  echo "Provide aws_key" >&2
  exit 201
fi
if [ -z "$3" ]; then
  echo "Provide aws_secret" >&2
  exit 201
fi

UBER_JAR=$1
UBER_JAR_NAME=$(basename $UBER_JAR)
AWS_KEY=$2
AWS_SECRET=$3
REGION=eu-west-1
KEY_PAIR_NAME=lvsl-ec2-dev
VPC_SUBNET_ID=subnet-11bc3866
NUM_NODES=20
NODE_TYPE=c3.8xlarge
S3_LOGS_PATH=s3n://lvsl-spark-logs-dub/

function die {
  echo ${1:-"Unknown error"} >&2
  exit 200
}

# Create an EMR cluster
CLUSTER_ID=$(aws emr create-cluster --name "Spark cluster" \
                    --bootstrap-action Path=s3://elasticmapreduce/bootstrap-actions ↩
                        /configure-hadoop,Args=[-y,yarn.nodemanager.resource.memory ↩
                        -mb=80000] \
                    --ami-version 3.8 \
                    --applications Name=Spark,Args=[-x,-l,DEBUG,-d,spark.driver. ↩
                        cores=20,-d,spark.driver.maxResultSize=6000m,-d,spark. ↩
                        driver.memory=15240m,-d,spark.yarn.am.memory=15240m,-d, ↩
                        spark.yarn.am.cores=20,-d,spark.dynamicAllocation.enabled= ↩
                        false,-d,spark.rdd.compress=true,-d,spark.task.cpus=2,-d, ↩
                        spark.akka.threads=10] Name=Ganglia \
                    --ec2-attributes KeyName=$KEY_PAIR_NAME \
                    --instance-type $NODE_TYPE \
                    --instance-count $NUM_NODES \
                    --use-default-roles \
                    --enable-debugging \
                    --log-uri $S3_LOGS_PATH \
                    --ec2-attributes SubnetId=$VPC_SUBNET_ID,KeyName=$KEY_PAIR_NAME ↩
                        \
                    --region $REGION \
                    --output text)

if [ $? != 0 ]; then
  die "Failed to start an EMR cluster."
fi

echo "Cluster created: $CLUSTER_ID. Initialisation..."
# Waiting for cluster to initialize
while [[ ! $(aws emr describe-cluster --cluster-id $CLUSTER_ID --region $REGION -- ↩
    query "Cluster.Status.State" --output text ) =~ ^(WAITING|TERM) ]]; do
```

```
  if [ $? != 0 ]; then
    die "Unable to get EMR cluster status"
  fi
  echo -n '.'
  sleep 5
done

if [[ $(aws emr describe-cluster --cluster-id $CLUSTER_ID --region $REGION --query " ↩
    Cluster.Status.State" --output text ) =~ ^TERM ]]; then
  die "Cluster failed to start."
fi

# Get master's public DNS name
MASTER_NODE=$(aws emr describe-cluster --cluster-id $CLUSTER_ID --region $REGION -- ↩
    query "Cluster.MasterPublicDnsName" --output text)
if [ -z $MASTER_NODE ]; then
  die "Unable to get the master's public DNS name"
fi

# Start the proxy in background. This used from Foxy Proxy plugin
(ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i "/Users/$USER/. ↩
    ssh/$KEY_PAIR_NAME.pem" -ND 8157 hadoop@$MASTER_NODE) &

echo ' '
echo "Cluster is ready: http://$MASTER_NODE:18080/"

scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -i "/Users/$USER/. ↩
    ssh/$KEY_PAIR_NAME.pem" $UBER_JAR hadoop@$MASTER_NODE:/home/hadoop/ ↩
    $UBER_JAR_NAME
if [ $? != 0 ]; then
  die "Unable to SCP uber JAR."
fi

# Add step to the cluster
SPARK_ARGS="[--class,com.github.lvsl.thesis.sparkimpl.App,/home/hadoop/ ↩
    $UBER_JAR_NAME,/home/hadoop/output,$AWS_KEY,$AWS_SECRET,3840]"
STEP_ID=$(aws emr add-steps --region $REGION \
                          --cluster-id $CLUSTER_ID \
                          --output text \
                          --steps Type=Spark,Name="Spark Program",ActionOnFailure= ↩
                              TERMINATE_JOB_FLOW,Args=$SPARK_ARGS)
if [ $? != 0 ]; then
  die "Unable to launch a step."
fi

STEP_ID=${STEP_ID#STEPIDS}

echo "Step added: $STEP_ID"

# Wait for step completion --query 'Step.Status.State'
while [[ ! $(aws emr describe-step --cluster-id $CLUSTER_ID --step-id $STEP_ID -- ↩
    region $REGION --query "Step.Status.State" --output text ) =~ ^(COMPLETED| ↩
    CANCELLED|FAILED) ]]; do
  echo -n '.'
  sleep 5
done
if [[ ! $(aws emr describe-step --cluster-id $CLUSTER_ID --step-id $STEP_ID --region ↩
     $REGION --query "Step.Status.State" --output text ) =~ ^COMPLETED ]]; then
 die "$CLUSTER_ID/$STEP_ID was not completed!"
fi

echo "Step completed. Type yes to terminate the cluster"

read
aws emr terminate-clusters --region $REGION --cluster-ids $CLUSTER_ID
if [ $? != 0 ]; then
  die "Unable to terminate the cluster."
fi
kill -TERM $(jobs -p)
exit 0
```

Listing A.4: Script to run a Spark Job in EMR cluster

Appendix B

B.1 CloudWatch EC2 Instance Metrics

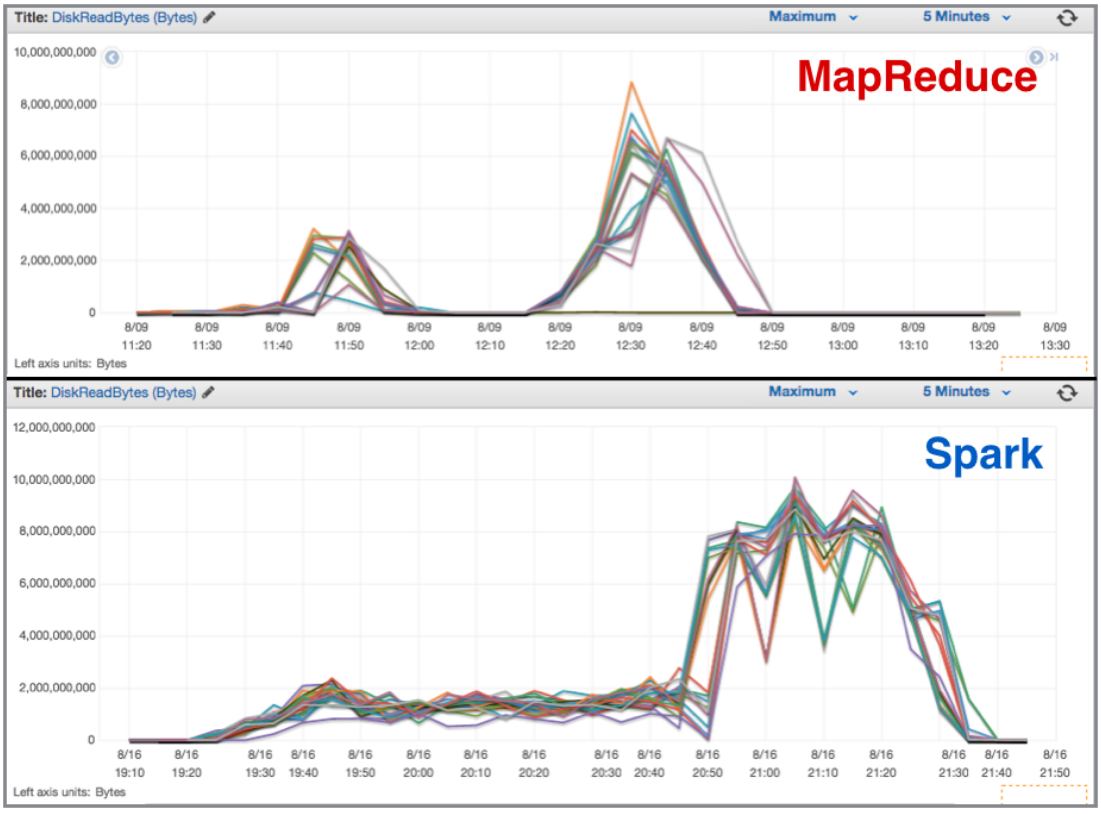

Figure B.1: EC2 Instance DiskReadBytes, Bytes

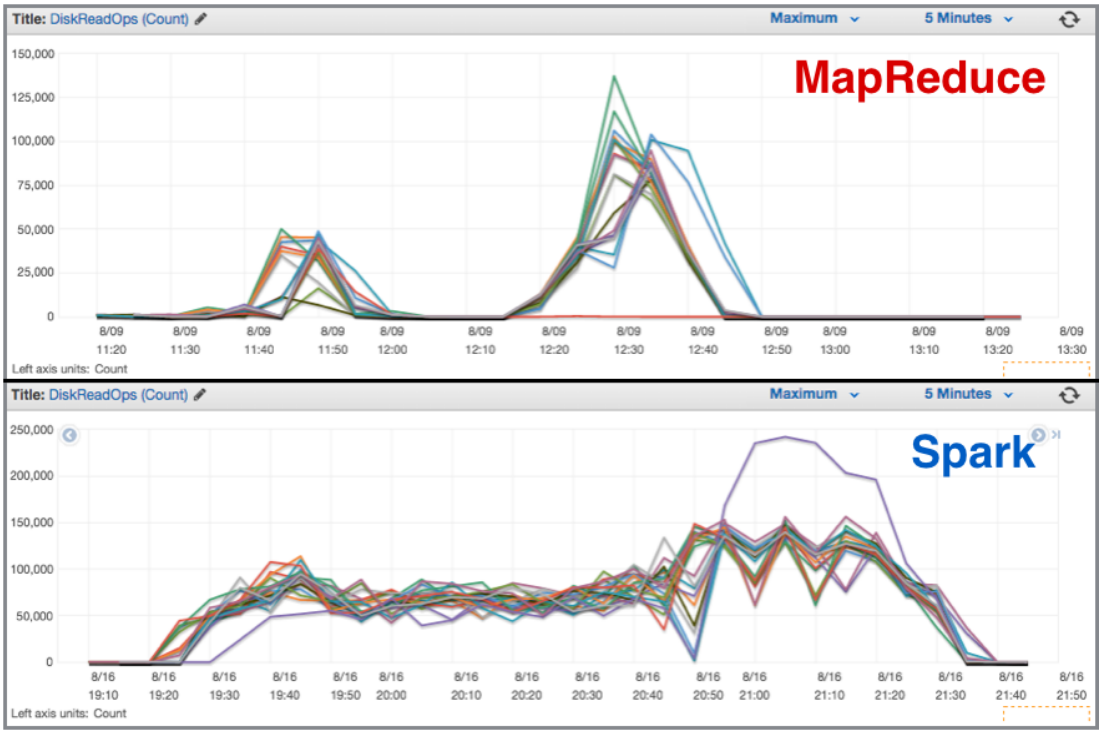

Figure B.2: EC2 Instance DiskReadOps, Number of IOPS

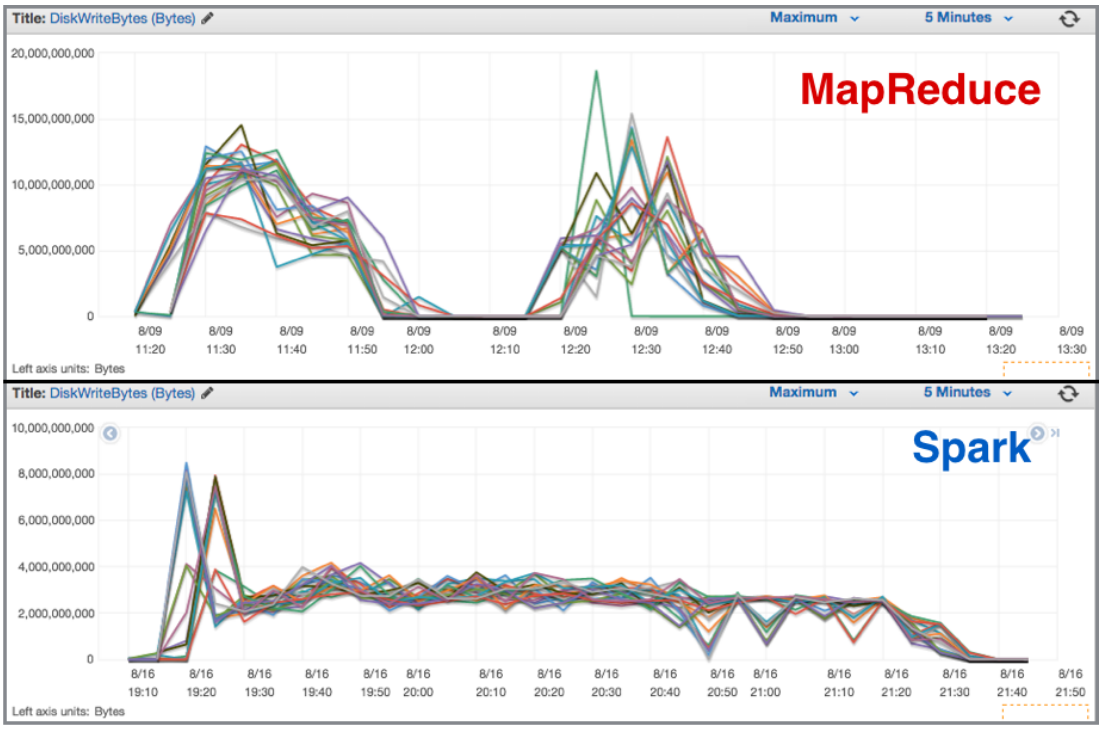

Figure B.3: EC2 Instance DiskWriteBytes, Bytes

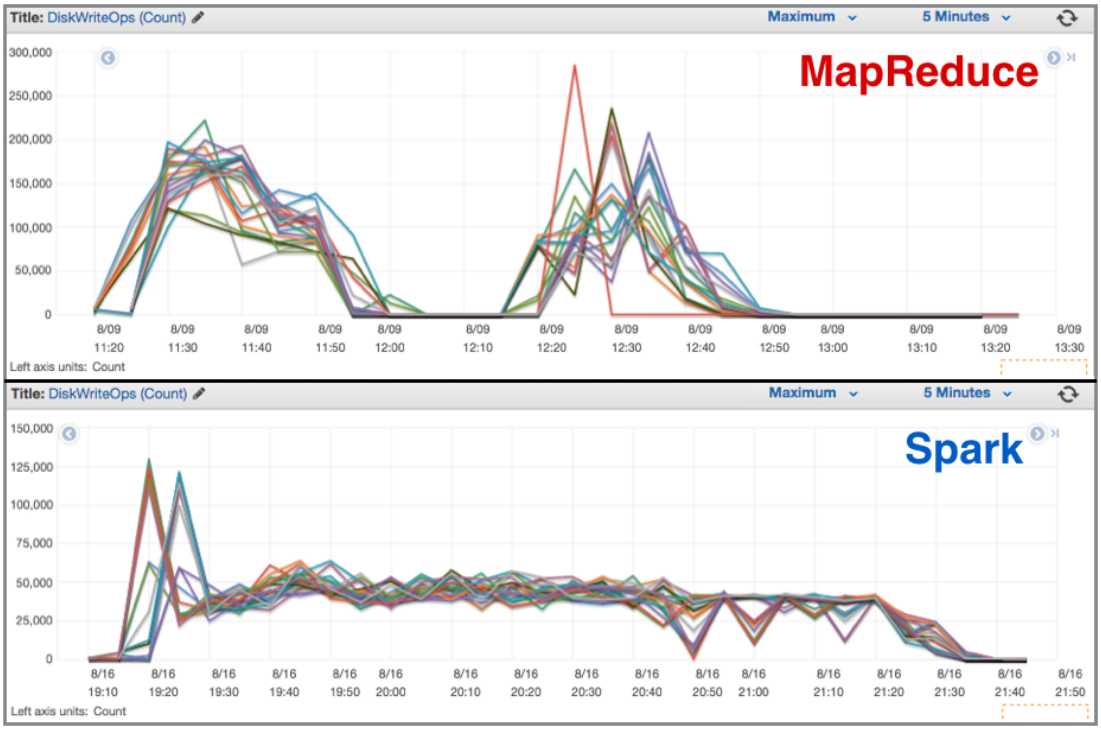

Figure B.4: EC2 Instance DiskWriteOps, Number of IOPS

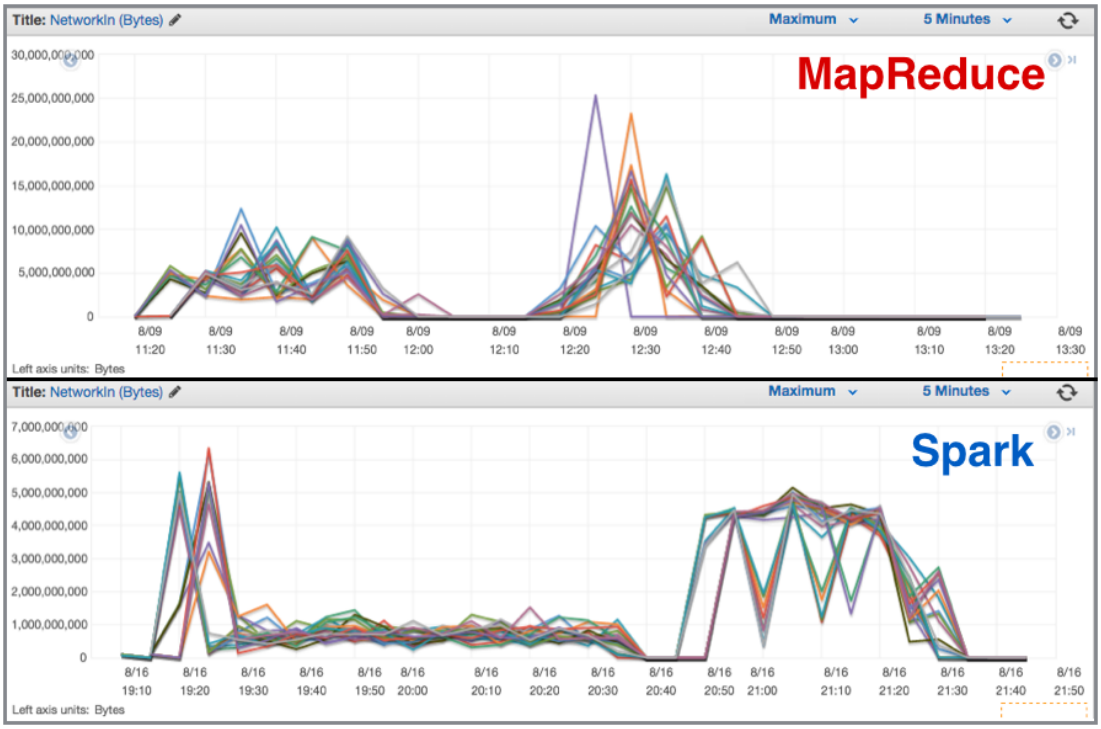

Figure B.5: EC2 Instance NetworkIn, Bytes

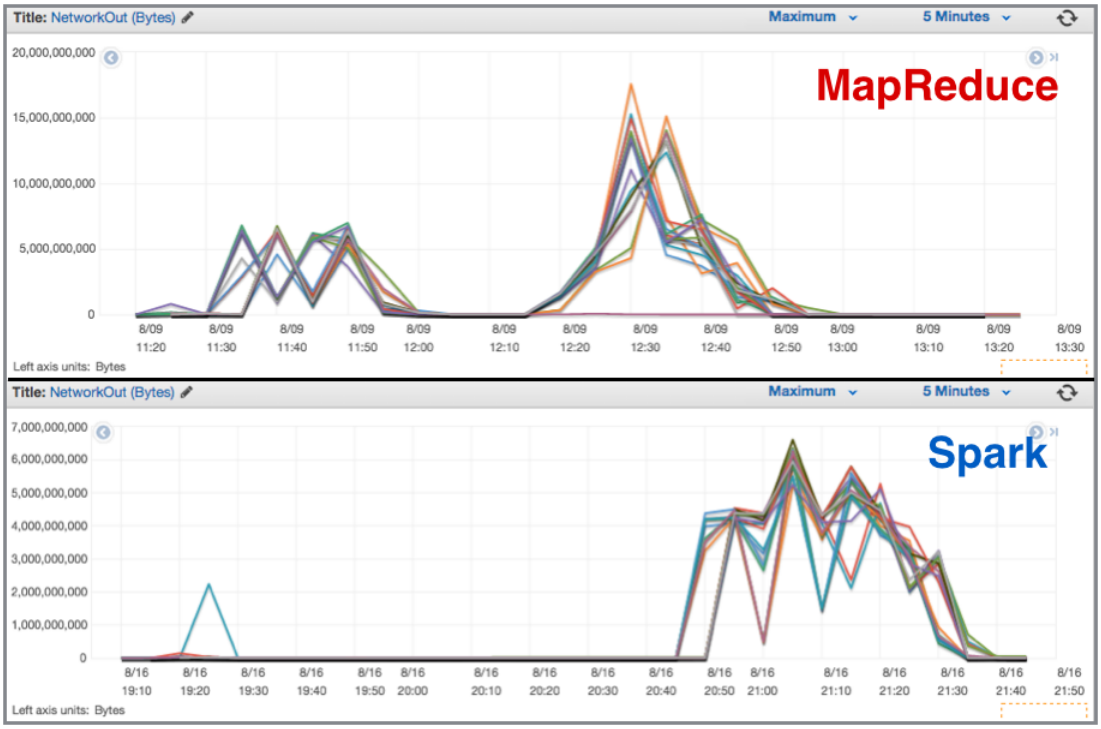

Figure B.6: EC2 Instance NetworkOut, Bytes

B.2 Spark Executor Metrics

Figure B.7 and Figure B.8 screenshots display various metrics produces by Spark tasks during the execution. *Stage 0* represents the mapping of the job, and *stage 1* the reducing part. Metrics have the following statistics: minimum, 25-th, 50-th(median), 75-th percentiles and maximum. This allows better observe the distribution of values for a particular metric.

Figure B.9 shows the detailed CPU Utilization (consume by userspace) metrics for a Spark cluster during the first stage: fetching journals from AWS S3 and parsing Avro.

Details for Stage 0

Total task time across all tasks: 994.5 h
Shuffle write: 1517.8 GB / 31445924416
Shuffle spill (memory): 9.3 TB
Shuffle spill (disk): 1451.3 GB

Show additional metrics

- *(De)select All*
- Scheduler Delay
- Task Deserialization Time
- Result Serialization Time
- Getting Result Time

Summary Metrics for 3861 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.8 s	9.1 min	14 min	22 min	51 min
Scheduler Delay	0 ms	9 ms	17 ms	53 ms	2 s
Task Deserialization Time	2 ms	4 ms	5 ms	11 ms	3 s
GC Time	0 ms	49 s	1.3 min	2.0 min	4.7 min
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	62 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	826.0 B / 5	218.6 MB / 4708212	356.7 MB / 7122492	581.6 MB / 11596892	1275.9 MB / 25476251
Shuffle spill (memory)	0.0 B	1446.6 MB	2.2 GB	3.5 GB	8.1 GB
Shuffle spill (disk)	0.0 B	202.7 MB	347.8 MB	556.3 MB	1261.8 MB

Figure B.7: Spark Stage 0 Metrics

Details for Stage 1

Total task time across all tasks: 438.6 h
Shuffle read: 1517.8 GB / 31445924416
Shuffle spill (memory): 8.5 TB
Shuffle spill (disk): 826.2 GB

Show additional metrics

- *(De)select All*
- Scheduler Delay
- Task Deserialization Time
- Shuffle Read Blocked Time
- Shuffle Remote Reads
- Result Serialization Time
- Getting Result Time

Summary Metrics for 2432 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	4.4 min	11 min	11 min	11 min	26 min
Scheduler Delay	3 ms	8 ms	16 ms	93 ms	0.5 s
Task Deserialization Time	1 ms	2 ms	2 ms	5 ms	0.1 s
GC Time	10 s	42 s	44 s	47 s	2.1 min
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	53 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Blocked Time	7 ms	25 ms	36 ms	51 ms	0.9 s
Shuffle Read Size / Records	613.7 MB / 11790622	626.0 MB / 12325459	629.9 MB / 12506355	636.0 MB / 12764891	1199.1 MB / 44553021
Shuffle Remote Reads	612.1 MB	624.3 MB	628.2 MB	634.3 MB	1197.3 MB
Shuffle spill (memory)	2.8 GB	3.4 GB	3.4 GB	3.4 GB	12.7 GB
Shuffle spill (disk)	267.5 MB	336.2 MB	338.3 MB	342.3 MB	911.2 MB

Figure B.8: Spark Stage 1 Metrics

Figure B.9: Spark Stage 1 Metrics

Appendix C

C.1 Spark JournalProcessor Code

This appendix contains full source code of a *Spark* based implementation of *JournalProcessor*.

```
// App.scala
package com.github.lvsl.thesis.sparkimpl

import java.io.File

import scala.Iterator
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable.ArrayBuffer

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

import com.amazonaws.services.s3.AmazonS3Client
import com.amazonaws.services.s3.model.ListObjectsRequest
import com.amazonaws.services.s3.model.ObjectListing
import com.amazonaws.services.s3.model.S3ObjectSummary
import com.github.lvsl.thesis.sparkimpl.data.AggObservation
import com.github.lvsl.thesis.sparkimpl.data.AggResult
import com.github.lvsl.thesis.sparkimpl.data.Observation
import com.tresata.spark.sorted.PairRDDFunctions.rddToSparkSortedPairRDDFunctions

object App {

  val PARTITIONS = 32 * 19 * 4 // 19 nodes of c3.8xlarge, 4 tasks per core

  val DATA_DIR = if (sys.env("USER") == "hadoop") {
    "/mnt/var"
  } else {
    "/tmp"
  }

  def main(args : Array[String]) {
    // ID for a job
    val runId = data.uuid

    // Configuration
    val conf = new SparkConf().setAppName("Lvsl thesis, run: %s" format runId)

    // Speed up with Kryo
    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    conf.set("spark.kryoserializer.buffer.max.mb", "1024")
    conf.set("spark.kryo.registrationRequired", "false")
    conf.registerKryoClasses(Array(
      classOf[data.AggObservation],
```

```
    classOf[data.AggResult],
    classOf[data.Observation]
  ))

  val sc = new SparkContext(conf)

  // XXX: for some reason s3: protocol does not work as well as * wildcard
  // use unions a hack to get RDD with multiple binary files
  sc.hadoopConfiguration.set("fs.s3n.awsAccessKeyId", args(1))
  sc.hadoopConfiguration.set("fs.s3n.awsSecretAccessKey", args(2))

  println("Spark Configuration:")
  sc.getConf.getAll.map(println)

  // List Bucket
  println("Listing journal bucket...")
  val s3client = new AmazonS3Client
  s3client.setEndpoint("https://s3-external-1.amazonaws.com")

  val listRequest = new ListObjectsRequest

  listRequest.withBucketName("lvsl-thesis-journals")

  var listResponse: ObjectListing = null
  val journals: ArrayBuffer[S3ObjectSummary] = new ArrayBuffer
  do {
    listResponse = s3client.listObjects(listRequest)
    journals ++= listResponse.getObjectSummaries
    listRequest.withMarker(listResponse.getNextMarker)
    println("Listed: %s" format journals.length)
  } while (listResponse.isTruncated)

  println("Done listing journals: %s" format journals.length)

  // Shuffle and take requested number of journals
  val numberOfJournals = {
    val n = args(3).toInt
    if (n == -1) {
      journals.length
    } else if (n > 0) {
      n
    } else {
      throw new IllegalArgumentException("Bad number of journals to process: %s"  ↩
          format n)
    }
  }
  // get bucket and key
  val shuffled = util.Random.shuffle(journals).take(numberOfJournals).map(
    summary => (summary.getBucketName, summary.getKey))

  println("Going to process: %s journals" format shuffled.length)

  // fetch the file and parse it
  val pairs = sc.parallelize(shuffled, shuffled.length).flatMap(x => {
    val (bucket, key) = x
    val localFile = new File("%s/%s" format (DATA_DIR, key))

    println("Going to fetch %s from %s" format (key, bucket))

    val tm = data.getS3TransferManager

    val download = tm.download(bucket, key, localFile)

    try {
      download.waitForCompletion
    } finally { tm.shutdownNow }

    println("Fetched %s from %s" format (key, bucket))
    try {
      data.readAvroJournal("%s/%s" format (bucket, key), new java.io. ↩
          FileInputStream(localFile))
```

```
    } finally {
      println("Removing temp file: %s" format localFile)
      val status = localFile.delete
      println("Removal status: %s" format status)
    }
  })

  // This will trigger shuffle
  val sorted = pairs.groupSort(PARTITIONS, Some(implicitly[Ordering[data. ↩
      Observation]]))
  val mapped = sorted.mapStreamByKey(data.aggregateObservations)


  val resultRDD = mapped.mapPartitionsWithIndex((index, items) => {
    val indexName = "%s_%s.index.txt.gz" format (runId, index)
    val chunkName = "%s_%s.chunk.txt.gz" format (runId, index)

    val indexFilePath = "%s/%s" format (DATA_DIR, indexName)
    val chunkFilePath = "%s/%s" format (DATA_DIR, chunkName)

    val indexFile = new File(indexFilePath)
    val chunkFile = new File(chunkFilePath)

    val writer = data.gzipTextWriter(indexFilePath)
    val chunkWriter = data.gzipTextWriter(chunkFilePath)

    var counter = 0
    var pos: Long = 0
    items.foreach(o => {
      writer.append("%s %s %s %s" format (o._1, o._2.count, o._2.duplicates, pos))
      writer.newLine()
      counter += 1

      // write chunk
      o._2.values.foreach(ao => {
        chunkWriter.append("%d %s %e %e %e %e" format (
          ao.startTimestmp, ao.unit, ao.statistics.count, ao.statistics.sum, ao. ↩
              statistics.min, ao.statistics.max))
        chunkWriter.newLine()
        pos += 1
      })
    })

    writer.close()
    chunkWriter.close()

    // Copy files to S3. Assume default credentials somewhow set
    val tm = data.getS3TransferManager

    println("[%s] Starting Upload..." format index)

    val indexUpload = tm.upload("lvsl-spark-output-dub", "%s/%s.index.txt.gz"  ↩
        format (runId, index), indexFile)
    val chunkUpload = tm.upload("lvsl-spark-output-dub", "%s/%s.chunk.txt.gz"  ↩
        format (runId, index), chunkFile)

    try {
      indexUpload.waitForCompletion
      chunkUpload.waitForCompletion
    } finally {
      tm.shutdownNow
      indexFile.delete
      chunkFile.delete
    }

    println("[%s] Upload complete!" format index)

    Iterator((index, counter))
  }, true)

  // Print RDD structure
```

```
    println(resultRDD.toDebugString)

    resultRDD.collect.map(println)
  }
}
```

Listing C.1: Main Application Class

```
// package.scala
  package com.github.lvsl.thesis.sparkimpl

  import java.io.BufferedWriter
  import java.io.File
  import java.io.FileOutputStream
  import java.io.InputStream
  import java.io.OutputStreamWriter
  import java.nio.ByteBuffer
  import java.nio.charset.Charset
  import java.security.MessageDigest
  import java.util.Arrays
  import java.util.zip.GZIPOutputStream

  import scala.Iterator
  import scala.Vector
  import scala.collection.JavaConverters.asScalaIteratorConverter
  import scala.collection.SortedMap
  import scala.math.Ordering.Implicits.infixOrderingOps

  import org.apache.avro.Schema
  import org.apache.avro.file.DataFileStream
  import org.apache.avro.generic.GenericDatumReader
  import org.apache.avro.generic.GenericRecord
  import org.apache.commons.compress.archivers.tar.TarArchiveEntry
  import org.apache.commons.compress.archivers.tar.TarArchiveInputStream
  import org.joda.time.DateTime
  import org.joda.time.format.ISODateTimeFormat

  import com.amazonaws.ClientConfiguration
  import com.amazonaws.regions.Region
  import com.amazonaws.regions.Regions
  import com.amazonaws.services.s3.AmazonS3Client
  import com.amazonaws.services.s3.transfer.TransferManager


  package object data {

    import scala.math.Ordering.Implicits._
    import scala.collection.JavaConverters._


    val ONE_MINUTE_MILLIS = 60 * 1000

    val OBSERVATION_PATTERN = "^Datapoint\\[AccountId=(.*),MetricId=(.*),FQMI=(.*), ↩
        Timestamp=(.*),AggegationId=(.*),Count=(.*),Sum=(.*),Max=(.*),Min=(.*),Unit ↩
        =(.*)\\]$".r

    val avroSchemaParser = new Schema.Parser

    val avroDatapointSchema = avroSchemaParser.parse("""{
    |"namespace": "com.github.lvsl.avro",
    |"name": "AvroDatapoint",
    |"type": "record",
    |"fields": [
    |  {"name": "metricName", "type": [ "string", "null" ]},
    |  {"name": "metricId", "type": "long"},
    |  {"name": "aggregationId", "type": "int"},
    |  {"name": "unit", "type": "string"},
    |  {"name": "accountId", "type": "string"},
    |  {"name": "timestamp", "type": "long"},
    |  {"name": "putAtTimestamp", "type": "long"},
    |  {"name": "count", "type": "double"},
```

```
    |  {"name": "min", "type": "double"},
    |  {"name": "max", "type": "double"},
    |  {"name": "sum", "type": "double"},
    |  {"name": "dimensions", "type": [ { "type": "map", "values": "string" }, "null ↩
        " ]},
    |  {"name": "distribution", "type": [ "bytes", "null" ]}
    |]
    |}""".stripMargin)

    val avroDatapointReader = new GenericDatumReader[GenericRecord]( ↩
        avroDatapointSchema)


    /**
     * Statistics object
     */
    case class Statistics(
      count: Double,
      sum: Double,
      min: Double,
      max: Double) {

      val avg: Double = sum / count
    }

    /**
     * Observation object
     */
    case class Observation(
      statistics: Statistics,
      timestamp: Long,
      unit: String,
      aggToken: Int
      ) extends Ordered[Observation] {

      def compare(that: Observation): Int = {
        val a = (this.timestamp, this.aggToken)
        val b = (that.timestamp, that.aggToken)
        if (a == b) {
          0
        } else if (a < b) {
          -1
        } else {
          1
        }
      }
    }

    /**
     * Parse string into an Observation object
     */
    def parseObservation(s: String): (String, Observation) = {
      val OBSERVATION_PATTERN(_, _, metricId, isoTime, aggId, count, sum, max, min,  ↩
          unit) = s
      val stats = new Statistics(count.toDouble, sum.toDouble, min.toDouble, max. ↩
          toDouble)
      val dt: DateTime = ISODateTimeFormat.dateTimeParser().parseDateTime(isoTime)

      (metricId, new Observation(stats, dt.getMillis, unit, aggId.toInt))
    }

    /**
     * Generate index and chunk for partition, upload them to S3.
     * Return iterator with partition index and summary about partition
     */
    def storePartition(partitionIndex: Int, items: Iterator[(String, AggObservation) ↩
        ]): Iterator[(Int, String)] = {
      Iterator((partitionIndex, "Size of partition: " + items.size))
    }

    /**
```

```
 * Aggregation result
 */
case class AggResult(
  duplicates: Int,
  count: Int,
  values: Vector[AggObservation]
)

/**
 * Aggregate an iterator of Observations into series of AggObservations
 * Note: Observations must be sorted
 */
def aggregateObservations(obs: Iterator[Observation]): TraversableOnce[AggResult ↩
    ] = {
  if (obs.isEmpty) {
    throw new IllegalArgumentException("Got empty iterator")
  }

  var items: SortedMap[(String, Long), AggObservation] = SortedMap()

  var currentAggToken: Long = 0
  var prevTimestamp: Long = -1
  var duplicates = 0
  var count = 0
  while (obs.hasNext) {
    val o = obs.next
    if (o.timestamp < prevTimestamp) {
      throw new IllegalStateException("Observations are not sorted")
    } else {
      prevTimestamp = o.timestamp
    }
    if (o.aggToken != currentAggToken) {
      count += 1
      currentAggToken = o.aggToken
      val key = (o.unit, truncateMillis(o.timestamp))

      if (items.contains(key)) {
        items += (key -> items(key).aggregate(o))
      } else {
        val agg = AggObservation(key._2, key._1)
        agg.aggregate(o)
        items += (key -> agg)
      }
    } else {
      duplicates +=1
    }
  }
  Iterator(AggResult(duplicates, count, items.values.toVector))
}

/**
 * Truncate milliseconds to a second
 */
def truncateMillis(millis: Long): Long = millis - (millis % ONE_MINUTE_MILLIS)

/**
 * Aggregated observations into one minute buckets
 */
case class AggObservation(
  startTimestmp: Long,
  unit: String
  ) {

  var statistics: Statistics = null

  var total_observations = 0

  def aggregate(obs: Observation): AggObservation = {

    if (obs.unit != this.unit) {
      throw new IllegalArgumentException(
```

```
          "Attemp to aggregate observation with different unit")
      }

      if (obs.timestamp < this.startTimestmp ||
          obs.timestamp - this.startTimestmp > ONE_MINUTE_MILLIS) {
        throw new IllegalArgumentException(
          "Attempt to aggregate observation for wrong bucket")
      }

      statistics = if (statistics == null) {
        obs.statistics
      } else {
        new Statistics(
          statistics.count + obs.statistics.count,
          statistics.sum + obs.statistics.sum,
          math.min(statistics.min, obs.statistics.min),
          math.max(statistics.max, obs.statistics.max))
      }

      total_observations += 1

      this
    }

    override def toString(): String = {
      "AggObservation(" + startTimestmp + ", " + unit + ", " + total_observations + ↩
          ", " + statistics.toString() + ")"
    }
  }

  /**
   * Journal header reader
   *
   */
  def readJournalHeader(stream: InputStream): (Boolean, Int) = {
    // check the magic

    val OJML = Vector(0x4f, 0x4a, 0x4d, 0x4c)
    val magic = Vector(stream.read(), stream.read(), stream.read(), stream.read())

    // check version
    val (hi, lo) = (stream.read(), stream.read())
    (magic == OJML, (hi << 8) + lo)
  }

  /**
   * Compute canonical metricId
   */
  def computeCanonicalMetricId(accId: String, metricId: Long): String = {
    val hasher = MessageDigest.getInstance("SHA-1")
    hasher.update(accId.getBytes(Charset.forName("UTF-8")))

    val buff = ByteBuffer.allocate(8)
    buff.putLong(metricId)

    hasher.update(buff.array)

    val value = Arrays.copyOfRange(hasher.digest(), 0, 16)

    value.map("%02x" format _).mkString
  }

  /**
   * Parse Avro GenericRecord into tuple: (canonicalMetricId, Observation)
   */
  def parseAvroRecord(record: GenericRecord): (String, Observation) = {
    val canonicaMetricId = computeCanonicalMetricId(
      record.get("accountId").asInstanceOf[org.apache.avro.util.Utf8].toString,
      record.get("metricId").asInstanceOf[Long])

    val stats = Statistics(
```

```
      count=record.get("count").asInstanceOf[Double],
      sum=record.get("sum").asInstanceOf[Double],
      min=record.get("min").asInstanceOf[Double],
      max=record.get("max").asInstanceOf[Double])

    val ob = Observation(
      statistics=stats,
      timestamp=record.get("timestamp").asInstanceOf[Long],
      unit=record.get("unit").asInstanceOf[org.apache.avro.util.Utf8].toString,
      aggToken=record.get("aggregationId").asInstanceOf[Int]
    )

    (canonicaMetricId, ob)
  }

  /**
   * Read header of embedded journal
   */
  def readEmbeddedJournalHeader(entry: TarArchiveEntry, tarStream: ↩
      TarArchiveInputStream): Boolean = {
    val GOOD_JOURNAL_HEADER = (true, 7)
    val journalHeader = readJournalHeader(tarStream)
    if( journalHeader != GOOD_JOURNAL_HEADER) {
      throw new IllegalStateException("Bad embedded journal(%s[%s]): %s" format ( ↩
          entry.getName, entry.getSize, journalHeader))
    }
    true
  }

  /**
   * Read avro journals from a stream
   */
  def readAvroJournal(fileName: String, content: java.io.FileInputStream): ↩
      Iterator[(String, Observation)]= {

    if(readJournalHeader(content) != (true, 6)) {
      throw new IllegalStateException("Bad journal")
    }

    // read tar archive
    val tarArchive = new TarArchiveInputStream(content)
    for {
      index <- Iterator.from(1).takeWhile( x => {
        val entry = tarArchive.getNextTarEntry
        println(" === Bytes read from %s: %s [tar: %s] ===" format (fileName, ↩
            tarArchive.getBytesRead, entry.getName))
        entry != null && entry.getName != "MANIFEST" && readEmbeddedJournalHeader( ↩
            entry, tarArchive)
      })
      record <- (new DataFileStream[GenericRecord](tarArchive, avroDatapointReader) ↩
          ).iterator.asScala
    } yield {
      parseAvroRecord(record)
    }
  }

  /**
   * Gzipped text writer
   */
  def gzipTextWriter(filename: String): BufferedWriter = {
    val file = new File(filename)
    val outStream = new FileOutputStream(file, false)
    val zipStream = new GZIPOutputStream(outStream)
    val outWriter = new OutputStreamWriter(zipStream, "ASCII")
    new BufferedWriter(outWriter)
  }

  /**
   * Get UUID
   */
  def uuid = java.util.UUID.randomUUID.toString
```

```
  /**
   * Get configured TransferManager client
   *
   */
  def getS3TransferManager(): TransferManager = {
    val cliConfig = (new ClientConfiguration)
      .withMaxConnections(10)
      .withMaxErrorRetry(10)
      .withGzip(false)
      .withSocketTimeout(3000)
      .withConnectionTimeout(3000)
      .withTcpKeepAlive(true)
      .withConnectionTTL(3000) // 3 sec

    val s3Cli: AmazonS3Client = (new AmazonS3Client(cliConfig))
      .withRegion(Region.getRegion(Regions.US_EAST_1))

    new TransferManager(s3Cli)
  }
}
```

Listing C.2: Various utility code