Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 156

MapReduce to Couple a Bio-mechanical and a Systems-biological Simulation

Alexander Christoph Gessler

Course of Study: Informatik

Examiner: PD Dr. rer. nat. habil. Holger Schwarz

Supervisor: Dipl.-Inf. Peter Reimann

Commenced: July 12, 2014

Completed: December 12, 2014

CR-Classification: H.2.4, H.2.8, H.2.3, H.3.4, H.4.1

Kurzfassung

Workflow Technologien werden aktuell verstärkt eingesetzt in der Hoffnung, hierdurch komplexe wissenschaftliche Simulationsabläufe einfacher umsetzen zu können. Das Thema dieser Arbeit ist ein existierender Workflow, der eine multiskalare Simulation des Massenflusses im porösen menschlichen Knochenmaterial umsetzt. Diese Simulation besteht aus getrennten systembiologischen und biomechanischen Berechnungen, die durch weitere Datenverarbeitungsschritte miteinander verbunden sind. Der Workflow weist ein erhebliches Potenzial zur Parallelisierung auf, welches allerdings nur geringfügig genutzt wird. Wir untersuchen daher, inwieweit sich "Big Data"-Konzepte wie etwa MapReduce oder NoSQL-Datenbanksysteme auf den Workflow übertragen lassen.

Ein Prototyp des Workflows wird mithilfe des Apache Hadoop-Ökosystems zur Parallelisierung der Simulation entwickelt und mit einem von Hand parallelisierten zweiten Prototyp in Bezug auf Effizienz und Skalierbarkeit verglichen. NoSQL-Konzepte zum Speichern von Eingaben und Resultaten werden angewendet, hierbei liegt der Fokus auf HDFS, dem Hadoop File System, als schemalosem, verteiltem Dateisystem und MySQL Cluster als einem Hybriden aus klassischem Datenbanksystem und einem NoSQL-Ansatz.

Zuletzt wird der MapReduce-basierte Prototyp in die Workflow-Beschreibungssprache WS-BPEL übertragen, wobei das SIMPL-Rahmenwerk[RRS+11] und ein spezieller Web Service zur Anbindung an Hadoop zum Einsatz kommen. Wir zeigen die Einfachkeit des resultierenden Workflows und halten fest, dass der gewählte Ansatz nicht nur den Implementierungsaufwand für Workflows verringert, sondern es auch einfacher macht, sich größerem Datenaufkommen anzupassen.

Abstract

Recently, workflow technology has fostered the hope of the scientific community in that they could help complex scientific simulations to become easier to implement and maintain. The subject of this thesis is an existing workflow for a multi-scalar simulation which calculates the flux of porous mass in human bones. The simulation consists of separate systems-biological and bio-mechanical simulation steps coupled through additional data processing steps. The workflow exhibits a high potential for parallelism which is only used to a marginal degree. Thus we investigate whether "Big Data" concepts such as MapReduce or NoSQL can be integrated into the workflow.

A prototype of the workflow is developed using the Apache Hadoop ecosystem to parallelize the simulation and this prototype compared against a hand-parallelized baseline prototype in terms of performance and scalability. NoSQL concepts for storing inputs and results are utilized with an emphasis on HDFS, the Hadoop File System, as a schemaless distributed file

system and MySQL Cluster as an intermediary between a classic database system and a NoSQL system.

Lastly, the MapReduce-based prototype is implemented in the WS-BPEL workflow language using the SIMPL[RRS+11] framework and a custom Web Service to access Hadoop functionality. We show the simplicity of the resulting workflow model and argue that the approach greatly decreases implementation effort and at the same time enables simulations to scale to very large data volumes at ease.

Contents

1 Introduction 9
   1.1 Task and Motivation 9

2 Foundations 11
   2.1 Service-oriented Architecture 11
   2.2 Workflows 12
   2.3 Bone Simulation 14
   2.4 MapReduce and Apache Hadoop 16
   2.5 NoSQL 22

3 Cluster Setup 27
   3.1 Computing Clusters: Overview 27
   3.2 OpenStack 28
   3.3 Apache Ambari 28
   3.4 Setup 30

4 Concept 35
   4.1 Data Flow of the Coupling Workflow 35
   4.2 Data Size 38
   4.3 Appropriate Use of MapReduce 39
   4.4 Parallelization of the Workflow using MapReduce 41
   4.5 MapReduce as a Service 43

5 Implementation 47
   5.1 MapReduce-WSI 47
   5.2 MapReduce Prototype 51
   5.3 Baseline Prototype 59

6 Evaluation 63
   6.1 Generating Test Data 63
   6.2 Challenges with Hadoop 64
   6.3 Challenges due to Virtualization 65
   6.4 Virtual Cluster Configuration 66
   6.5 Optimizing MySQL Performance 72
   6.6 Measuring MapReduce vs. Baseline 75

7 Workflow Integration 85
7.1 Formulating the MapReduce prototype using WS-BPEL 85

8 Conclusion 87
8.1 Future Work 88

A Appendix A - WSDL for MapReduce-WSI 91

B Appendix B - Code to Run and Measure the MapReduce prototype 93

C Appendix C - Code to Generate Artificial Test Data 97

Bibliography 103

List of Figures

2.1 SOA Triangle . . . 11
2.2 Sample WS-BPEL workflow displayed in a graphical designer . . . 13
2.3 Workflow Overview . . . 15
2.4 Hadoop Architecture . . . 18

3.1 Ambari Status Dashboard . . . 29
3.2 Ambari YARN Metrics Dashboard . . . 30
3.3 Cluster Topology . . . 31

4.1 Original Coupling Workflow: Data and Control Flow . . . 35
4.2 Data and Control Flow of the MapReduce prototype . . . 44

5.1 YARN Applications Web UI . . . 58
5.2 MapReduce progress Web UI running the first MR stage . . . 59

6.1 Physical Topology of the Virtual Cluster . . . 68
6.2 MapReduce vs. Baseline Prototype Evaluation Results . . . 79
6.3 Results with Respect to Input Data Size, Parallelism #32 . . . 79
6.4 Comparison of the Prototypes, Parallelism #32 . . . 80
6.5 Results with Respect to Input Data Size (Normalized), Parallelism #32 . . . 80
6.6 Results with Respect to Output Data Size (Normalized), Parallelism #32 . . . 81
6.7 Cluster CPU Usage with Sub-optimal reducer Scheduling . . . 83

A.1 WSDL Interface Description for MapReduce-WSI . . . 91

List of Tables

6.1 Measurements: *aggregation_time* (seconds) . . . 78
6.2 Measurements: *octave_time* (seconds) . . . 78
6.3 Measurements: *total_time* (seconds) . . . 78

List of Listings

2.1 AnagramFilter mapper script 19
2.2 AnagramFilter reducer script 19

4.1 SQL schema of the PANDAS output table 36
4.2 SQL query to calculate aggregate variable values from `pandas_output` 37
4.3 SQL schema of the Octave output table 38

5.1 Sqoop import statement 51
5.2 AverageTimesteps mapper script 53
5.3 AverageTimesteps reducer script 54
5.4 OctaveCalc mapper script 56
5.5 OctaveCalc reducer script 57
5.6 Query to extract a worker's subset of the input data 60

6.1 MySQL Cluster configuration (`ndbmtd`) 71

1 Introduction

Scientific simulations have grown in scale and now oftentimes require massive computational resources and complex software provisioning to produce and consume large volumes of data. This makes it harder to achieve reproducible, data-driven research, which is the classic goal of empirical science. Workflow technology is an industry-proven approach to structure sequences or flows of activities using high-level descriptions [LR00]. In the scientific community, workflow technology is increasingly used to make simulations easier to formulate and maintain [TDGS06].

The (very broad) term *Big Data* refers to technologies that enable highly parallel processing and storage of large volumes of data. An example is the MapReduce paradigm, which formulates computations such that they can be automatically distributed on clusters of commodity hardware[DG08]. Another example is the NoSQL paradigm, which emphasizes data stores that endow their data with less structure and provide weaker guarantees than classic SQL-based systems, but instead possess the ability to scale to dynamic workloads and to be highly fault-resilient.

1.1 Task and Motivation

In the scope of this thesis, the applicability of various Big Data concepts and tools to scientific workflows and scientific simulations is discussed. The presentation and evaluation is based on a specific example of a complex scientific workflow that has been subject of previous publications by the University of Stuttgart's Institute for Parallel and Distributed Systems, such as presented by Reimann et al. [RSM14b], [RSM14a]. This workflow simulates the flux of mass within the pourous material of human bones during habitual daily activity and consists of complex simulation and data transformation steps. The two key components are a relatively coarse-grained bio-mechnical simulation (which is carried out using the University of Stuttgart's Pandas FEM solver[pan]) and a more fine-grained, systems-biological simulation (which is implemented via GNU Octave[oct]). Both simulations are integrated together in a so-called coupling workflow.

However, this simulation exhibits a large amount of parallelism which is difficult to utilize from the used workflow language, at least not without increasing the complexity of the workflow

models. Also, the volume of the data processed during the workflow is large such that NoSQL approaches should be considered to store data.

A motivation behind this thesis is to develop a mechanism through which Apache Hadoop, an ecosystem that enables large-scale parallel calculations and data storage, can be utilized from within a scientific workflow. A goal is to develop such a binding and to evaluate its performance with respect to more classic approaches to parallelize the bone simulation. More importantly however, this thesis investigates whether such a system can reduce the complexity and cost of maintenance and increase flexibility of the resulting scientific workflows with respect to growing demands.

While using Apache Hadoop is the focus of the thesis, different models, such as PACT, are briefly contrasted with Hadoop[AHM+10].

Structure

This thesis is structured as follows:

Chapter 2 – Foundations: This chapter describes relevant foundations for this thesis. This includes a look at MapReduce as a paradigm, and its implementation in Apache Hadoop. The bone simulation that this thesis is centered around is briefly described.

Chapter 3 – Cluster Setup: This chapter describes how we create and monitor Hadoop clusters. It also defines the overall computing environment used for this thesis.

Chapter 4 – Concept: This chapter describes a concept for how the entire Coupling Workflow, encompassing both the systems-biological and the bio-mechnical simulation, is mapped to the MapReduce paradigm.

Chapter 5 – Implementation: This chapter describes the specific prototypical implementation of the concept. It describes important challenges faced during implementation, as well as shortcomings of the resulting Java prototype. We also describe how a second, baseline prototype that does not rely on any MapReduce concepts is built.

Chapter 6 – Evaluation: This chapter describes the setting and results of the evaluation of the prototype in comparison with the baseline version. Criteria of interest are performance and scalability in comparison to the baseline.

Chapter 8 – Conclusion Sums up the work done, emphasizes key results and concludes with an outlook into potential further work.

2 Foundations

This chapter aims at giving a brief overview of topics that are relevant for this thesis. In particular, the concept of a Service-oriented architecture (SOA) is introduced as well as the Workflow technology that gives rise to the topic in the first place. The bone simulation as a concrete example of a complex simulation workflow is characterized and previous work is summarized. Furthermore, the MapReduce and "NoSQL" paradigms are explained.

2.1 Service-oriented Architecture

A Service-oriented architecture (SOA) is a software architecture style that emphasizes the use of loosely coupled, self-contained services that offer functionality to applications.

There are many possible definitions of the term SOA. As [CSF06] puts it, SOA ..

> ... describes how service consumers and service providers can be decoupled via discovery mechanisms resulting in loosely coupled systems [..] Implementing a service-oriented architecture means to deal with heterogeneity and interoperability concerns.

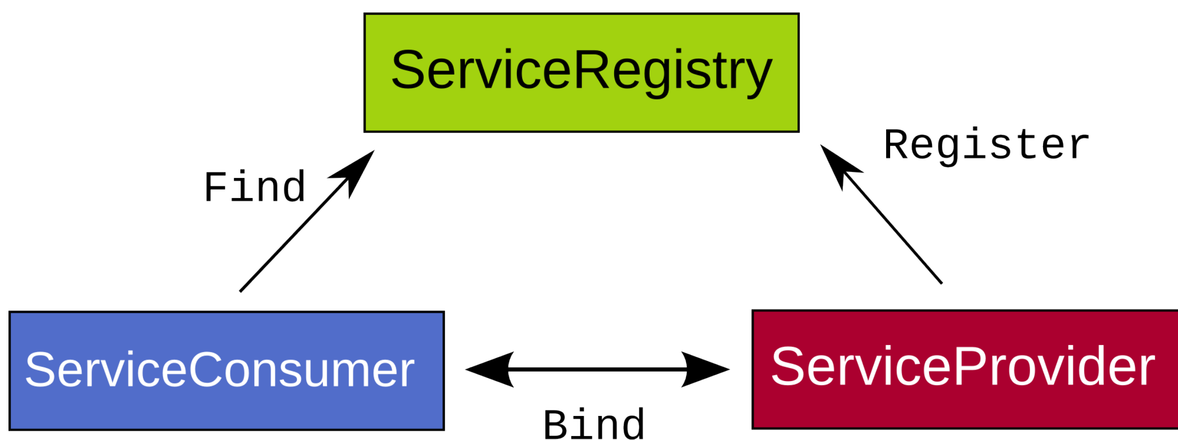


Figure 2.1: SOA Triangle

Loosely coupled means in this context that an application is not dependent upon a particular set of services or service vendors. Any service can be easily exchanged against another service that offers the same public API. The so-called SOA triangle in Figure 2.1 is used to classify different roles in a SOA: If a party offers a service, they act as a *service provider* and publish an abstract description of their service to a *service registry*. If an application requires a certains service, it acts as a *service consumer*. It first asks the service registry for a service or list of services that best match its requirements (i. e., kind of service offered, SLA, price & billing). The consumer then requests a description of the chosen service from the the service provider that offers it. This description enables the consumer to invoke the service offered.

2.1.1 Web Services

Web Services are a common, standardized implementation of a SOA. A Web Service can be described by an interface description formulated in WSDL, the *Web Service Definition Language*[1]. To exchange messages between the consumer of a Web Service and a Web Service, SOAP, the *Simple Object Access Protocol*[2], is used. Both WSDL and SOAP are XML-based.

Web Services are governed by a family of interrelated standards that are usually prefixed by *WS-*. Standards that are relevant to this thesis are

- *WS-Adressing*[3], which allows consumers and services to associate a conversation context (session) with all messages they exchange, and
- *WS-BPEL*[4] for describing workflows. WS-BPEL is further described in section 2.2.1.

In Java, the JAX-WS[5] framework provides a reference implementation of Web Services as part of the Java Standard Edition (since 1.6) and the Java Enterprise Edition (since 1.5).

2.2 Workflows

A Workflow is a set of activities in well-defined order. By grouping activities in workflows, they can easily be re-used as part of larger workflows (Composability). Workflows technology has been advanced by IBM, Microsoft and others (for example through the aforementioned WS-BPEL [OAS07]), for formulating compless business processes. In the research community, workflow technology fosters the hope of making scientific simulations more reproducible, more reliable to execute and easier to maintain [GSK+11].

[1] http://www.w3.org/TR/wsdl20/
[2] http://www.w3.org/TR/soap/
[3] http://www.w3.org/Submission/ws-addressing/
[4] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[5] https://jax-ws.java.net/

2.2.1 WS-BPEL

The WS-Business Process Execution Language (WS-BPEL) is a language to describe workflows while integrating Web Services technology. WS-BPEL is standardized by OASIS[6] in multiple versions, 2.0 being the latest as of 2014 [OAS07]. Workflows that are formulated in WS-BPEL can invoke Web Services as embedded activies, and the entire workflow itself becomes a Web Service for use by others.

WS-BPEL pre-defines a variety of activity types that resemble classical programming language constructs, including activities for flow controls (conditional execution, *forEach* activity to handle looping and a *flow* activity to execute workflow sub-activities in parallel) and a mechanism to declare and assign variables for maintaining state during the workflow. Additionally, the borrowed concept of a *Scope* provides isolation and fault-handling to parts of a Workflow.

Figure 2.2 shows a simple sample WS-BPEL workflow displayed in the graphical Eclipse BPEL Designer[7]. The workflow first waits to be invoked (*Receive*). It then uses a ForEach-activity to run several iterations of an embedded activity (*Invoke*). Upon completion, it sends a *Reply* to the original requestor.

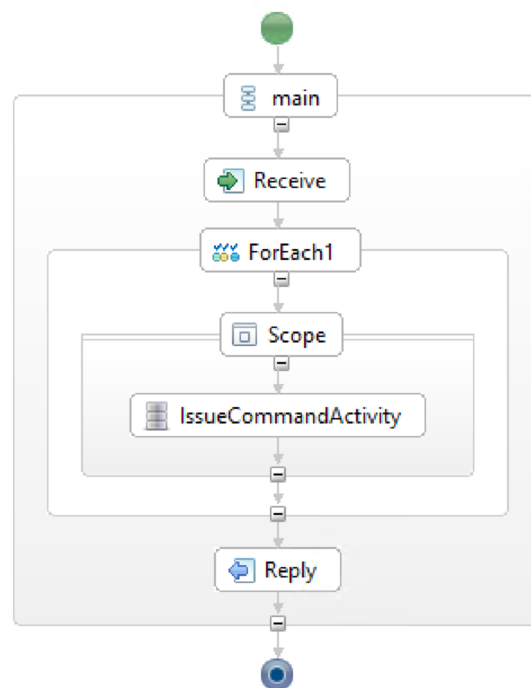


Figure 2.2: Sample WS-BPEL workflow displayed in a graphical designer

[6]Organization for the Advancement of Structured Information Standards, https://www.oasis-open.org/
[7]http://www.eclipse.org/bpel/

2.2.2 SIMPL

The SIMPL framework has been presented by Reimann et al. in [RRS+11] as a pattern-based approach to simplify data provisioning for scientific simulation workflows. The authors describe the problem they set to solve as follows:

> Today, data exchange with and between simulation applications is mainly accomplished in a file-style manner. These files show proprietary formats and have to be transformed according to the specific needs of simulation applications. Lots of effort has to be spent to find appropriate data sources and to specify and implement data transformations.

As a foundation, the SIMPL framework extends BPEL with activity types for data management operations. An example is the *IssueCommand* activity, which encapsulates a target-specific command (i.e, a SQL query or a shell command) and the *RetrieveData* respectively *WriteDataBack* activities which provide read/write access to a data source (such as a SQL-based database table or a file).

Furthermore, SIMPL provides abstract “template” activities, called patterns. The patterns are then transformed into the correct workflow activities depending on metadata and the context where the pattern is used, substituting any necessary data transformations. SIMPL organizes its patterns in a hierarchy with increasing abstraction and information density. An important pattern at the second-highest level of abstraction is the so-called *Simulation-oriented Data Provisioning Pattern* which allows scientists to bind inputs to workflow steps using concepts and terminology they are familiar with from simulation models. Specifically, an ontology describes the mathematical variables in a simulation and inputs are then referred to by their (mathematical) variable name [Boh14]. During pattern transformation, the ontology is used to substitute the correct data transfer and transformation pattern as a lower level of abstraction.

An example of such data transfer and transformation pattern is the *Join* pattern which represents the joining of two data sources on a specific key [RRS+11]. The concrete workflow activities in which an occurence of the pattern is transformed depends on the type and location of its operands and outputs. For example if both operands are SQL tables within the same database, a simple SQL `JOIN` is used to implement the pattern. If both operands refer to external data sources, separate activities to retrieve the data sources, join them and then potentially write the results into the destination data sink are generated.

2.3 Bone Simulation

The use case for this thesis is a simulation of changes to the structure of bones under the influence of external forces that occur habitually during daily activity of the relevant person.

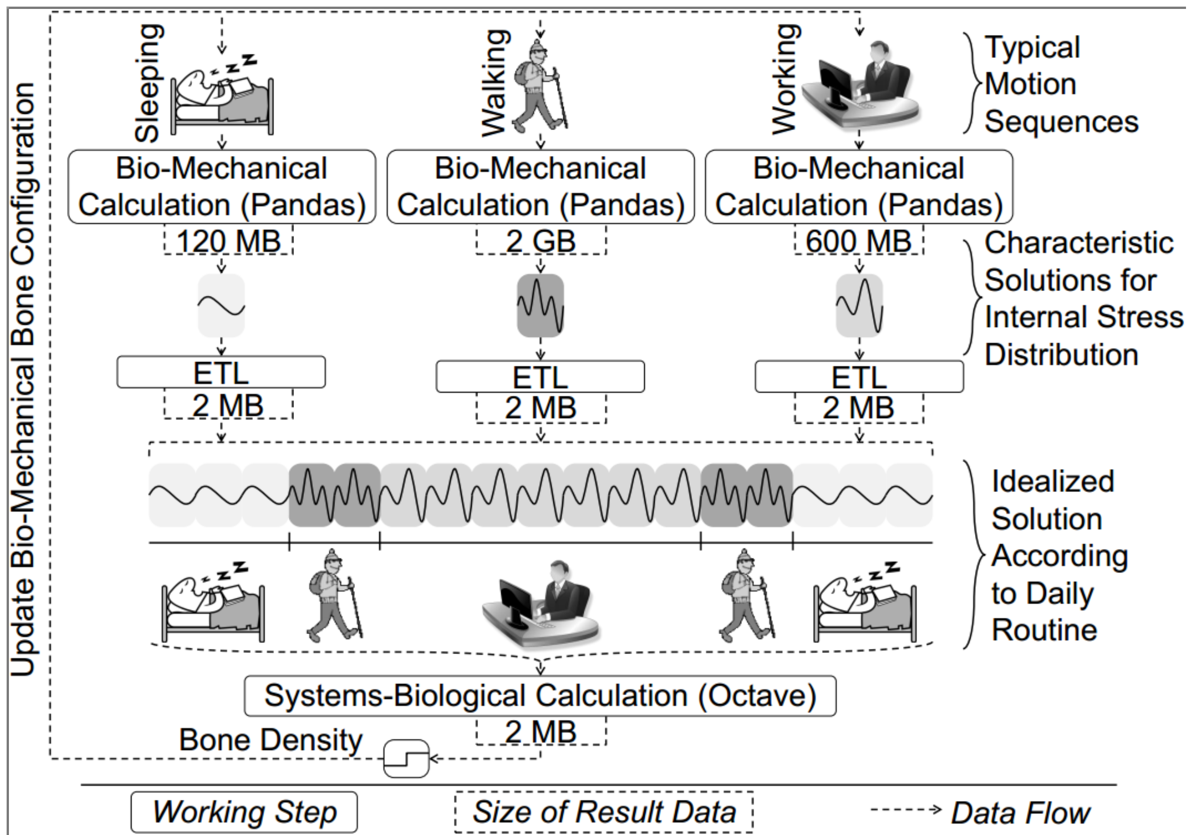


Figure 2.3: Workflow overview from Reimann et al.[RSM14a]

The scientific importance of this simulation is for example to better understand and improve healing processes after bone fractures [KSR+11]. The simulation is carried out in two steps.

1. On a bio-mechanical level, mass exchange between porous materials and liquids in the tissue of the bone is simulated. For this, the PANDAS framework, which has been developed at the University of Stuttgart, is used [pan].
2. On a systems-biological level, tissue changes on a cellular level are simulated using GNU Octave [oct].

The simulation always operates on a single bone.

Figure 2.3 gives a very high-level overview of the entire simulation based on Reiman et al. [RSM14a]. To model the forces that affect a bone during a day, multiple *Motion Sequences*, patterns of human activity that are characterized by their typical distribution of external forces on the bone, are identified. Examples of such Motion Sequences include Walking, Sleeping, Driving or Office Work. The bio-mechanical simulation (PANDAS) generates a characteristic solution to the distribution of forces inside a bone under each activity. The systems-biological

Octave simulation then puts these results together according to a *Daily Routine* and uses them to simulate tissue changes inside the bone over the course of one day. The system is designed to run over multiple, possibly different Daily Routines: the output of the Octave simulation for one Daily Routine produces new initial conditions for simulating the next bio-mechnical cycle with PANDAS. Initial conditions are pre-defined for the first cycle.

Both simulations have been presented as separate workflows that are coupled together by Reimann et al. [RSM14b]. Several other works by students at the University of Stuttgart have also worked on this topic area. Andreas Bohrn developed and presented implementations of the two workflows [Boh14]. The bio-mechnical simulation using PANDAS has already been abstracted into a JAX-WS-based Web Service by Raymond Dormien as part of his thesis [Dor11]. Lastly, Victor Riempp has engineered the *Coupling Workflow*, which encompasses the entire simulation [Rie14]. All workflows are based on the Workflow Patterns provided by the SIMPL framework (see Section 2.2.2).

The focus of this thesis is less on the actual simulation, than on exemplifying how a complex, large simulation can be parallelized using MapReduce and how this results in less effort than with manually implemented parallelization. This thesis is also not too bothered with the workflow technology per se: if our solutions utilize SOA principles, they can be easily integrated into any existing workflow and in particular into the SIMPL framework.

2.4 MapReduce and Apache Hadoop

This section gives a brief overview of the MapReduce paradigm and one of its most popular Open Source implementations, the *Apache Hadoop* framework. The MapReduce programs developed for this thesis will run on Hadoop.

2.4.1 The MapReduce paradigm

MapReduce is a programming paradigm introduced by Google in 2004 for performing large-scale, highly parallel batch computations on clusters [DG08]. Instead of burdening the programmer with having to manually parallelize their code, MapReduce formulates computations such that they can be automatically parallelized.

Conceptually, a set of input (key, value) pairs is transformed into a set of output (key, value) pairs. A MapReduce program consists of two stages, the `Map` first and the `Reduce` stage second. The programmer specifies a user-defined function for each of the stages. Each function is required to have no side effects and to only depend on its own inputs.

The `Map` stage independently takes each input (key, value) pair and maps it to zero or more (key, value) pairs using the user-defined mapper function. The MapReduce framework then

collects all pairs with the same key and groups them together. This intermediate step is known as `Shuffling`. For each group of pairs with the same key, the `Reduce` stage independently invokes the user-defined reducer function. The reducer function emits a new set of key-value pairs for the group.

Each stage allows for parallelization at a granularity down to each invocation of the mapper respectively reducer function. If a computation does not need a certain stage, it can supply the identity function to effectively disable it.

In Google's original paper, their implementation is able to distribute the workload across thousands of machines running in a computing cluster on commodity hardware [DG08]. The inputs and outputs for a MapReduce program often spread across thousands of files, called `shards`, that are stored in GFS, the company's distributed file system. Alternatively, databases such as BigTable are in use [CDG+08].

More recently, FlumeJava has been developed as a higher-level approach, using MapReduce as building block [CRP+10].

2.4.2 Apache Hadoop

The Apache Hadoop framework[8] is an Open Source implementation of various large-scale data processing tools, one of which is MapReduce. Now a top-level Apache Foundation project, it was founded by *Yahoo!* engineers in 2006, building on top of their previous work on Apache Lucene[9], a scalable full-text search engine, and the Nutch project[10]. It runs on clusters with commodity hardware and scales up to thousands of machines. Hadoop is in active use by many companies[11], including Yahoo! and *Facebook*. It has also become the preferred environment for academic work on large-scale data analysis, and it also forms the foundation for this thesis.

Hadoop is almost entirely based on the Java platform. It has three architectural layers as shown in Figure 2.4.

[8] http://hadoop.apache.org/
[9] http://lucene.apache.org/core/
[10] http://nutch.apache.org/
[11] https://wiki.apache.org/hadoop/PoweredBy

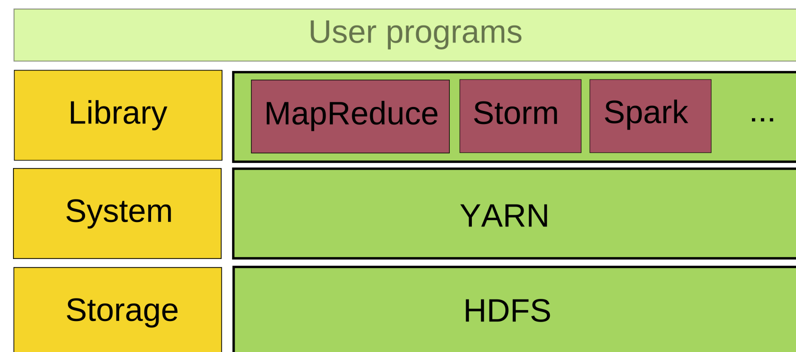


Figure 2.4: Hadoop Architecture (adapted from Hortonworks[12])

- `HDFS`[13], the *Hadoop Distributed File System*, is modelled after Google's GFS [GGL03]. It distributes files across available resources and offers fault-tolerance by storing multiple replicas of the data.
- `YARN`[14], or *Yet Another Resource Negotiator*, is a scheduling system for cluster resources. YARN keeps track of available resources (CPU cores, local storage, HDFS storage, RAM) in the cluster and allocates them to users. This allows a single Hadoop cluster to be used by multiple computations. In analogy to a single PC, YARN is the Operating System on which user code is executed.
- At a library/application level there are realizations of different data processing paradigms (i. e., MapReduce vs. realtime stream processing), which can all co-exist in a cluster.

Hadoop has grown into a large ecosystem and it integrates well with other Apache projects, such as Apache Cassandra (a column-oriented NoSQL database)[15] or Apache HBase (a BigTable-inspired distributed database)[16].

2.4.3 MapReduce on Hadoop

MapReduce on Hadoop closely follows Google's original paper [DG08].

Mapper and reducer function are normally written in Java, but there is also a special *Streaming Mode* in which an arbitrary program (i. e., a Python script or a C++ binary) is executed on each input record, using `STDIN` and `STDOUT` for IO [Apa13].

[12]http://hortonworks.com/hadoop/yarn/
[13]http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[14]http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[15]http://cassandra.apache.org/
[16]http://hbase.apache.org/

As a concrete non-trivial example, consider we are given a list of words and we want to remove as few words as possible such that the output list contains no anagrams. A simple rule is that for any pair of words that are anagrams of each other, one must be removed (which one does not matter and the solution is not unique).

Suitable Streaming Mode mapper and reducer scripts formulated in Python are shown in Listing 2.1 and 2.2. Some extra complexity to the otherwise easy Java-based MapReduce implementation is that Streaming Mode reducers get a continuous stream of (key, value) pairs sorted by key; so they have to detect when the key changes and act accordingly. In Python in particular, the `itertools.groupby` library function greatly simplifies this task and will be used in the remainder of this thesis.

Listing 2.1 AnagramFilter mapper script

```
#!/usr/bin/env python
import sys

# Map a sorted version of each word to the word itself.
# i.e., an input of "ananas" becomes ("aaanns", "ananas")
for line in sys.stdin:
  for word in line.strip().split():
    sorted_word = ''.join(sorted(word))
    print '%s\t%s' % (sorted_word, word)
```

Listing 2.2 AnagramFilter reducer script

```
#!/usr/bin/env python
import sys

last_sorted_word = None

# For each key emit the first value and skip all others.
# A more Pythonic approach uses itertools.groupby()
for line in sys.stdin:
   sorted_word, word = line.split('\t', 1)
   if sorted_word == last_sorted_word:
      continue

    last_sorted_word = sorted_word
    print word
```

For example, say the input to the MapReduce job is the following words in a text file, each word on its own line:

```
ananas
coconut
```

```
anasan
asanan
```

Then the output of the mapper stage is the following pairs, in no particular order:

```
aaanns ananas
aaanns anasan
ccnootu coconut
aaanns asanan
```

Note that the result would be the same if instead of invoking the mapper script once on the entire input, multiple instances of it were run on subsets of the input and results merged. More than four mapper script instances cannot run in this example since there are only four input words. After shuffling, the input to the reducer stage is the following list of pairs, which is now ordered by key (not by value though):

```
aaanns ananas
aaanns asanan
aaanns anasan
ccnootu coconut
```

Since `ananas, anasan, asanan` are all anagrams of each other, the two latter are discarded by the reducer script. Thus only two words are printed:

```
ananas
coconut
```

Note that the result would be the same if instead of invoking the reducer script once on all the pairs, two instances of the reducer script would be invoked on `(aaanns ananas, aaanns anasan, aaanns asanan)` and `(ccnootu coconut)` respectively and results merged. More than two reducer script instances cannot run in this example since there are only two unique keys emitted by the map stage.

Historically, Hadoop's initial MapReduce implementation (dubbed `mapred`) had been implemented on Hadoop directly. With Hadoop 0.23 (released in 2013[17]), the current implementation of MapReduce (dubbed `mapreduce2`) now runs on top of `YARN` where it takes advantage of its advanced scheduling and resource management[18].

[17]http://hadoop.apache.org/docs/r0.23.10/
[18]http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

2.4.4 MapReduce Fault Tolerance

One property of the MapReduce paradigm is that it allows for Fault Tolerance: mapper and reducer functions are *idempotent* by design, they can be re-executed without affecting the result.

Hardware failure is a normal, every-day scenario in a cluster that consists of hundreds or thousands of nodes. Hadoop detects node failures and re-executes the corresponding tasks on other nodes. If very few partitions of the input take significantly longer to process than all other partitions, the workers corresponding to these partitions are called *Stragglers*, a term coined as early as in the original Google paper [DG08]. As Zaharia et al. point out, Stragglers can be non-permanent: it could simply be a node in a heterogenous cluster that is faster than other nodes [ZKJ+08]. To mitigate such effects, Hadoop employs techniques such as *Speculative Execution* that re-start the same task on multiple nodes if one node is suspected to be slow. Once the first node finishes, all other replicas of the task are discarded[19].

2.4.5 Other data processing tools on top of Hadoop

Other data processing tools that run on Hadoop's YARN layer are:

- `Apache Spark`[20] is an in-memory execution engine for computations that are represented as a Directed Acyclic Graph (DAG), but also supports certain iterative (cyclic) flows. This makes Spark great for Machine Learning, where oftentimes models are iteratively trained before they are used for making predictions.
- `Apache Storm`[21] is for processing streams of data in realtime.

There is also a variety of higher-level (query) tools that build on top of MapReduce, giving users more expressive languages for analyzing large amounts of data, in particular for ad-hoc tasks.

- `Apache Pig` features a declarative high-level language, called *Pig Latin*. PigLatin is transparently compiled into lower-level MapReduce programs. Pig was originally developed by Yahoo![GNC+09].
- `Apache Hive` is an interactive query system that supports SQL-like queries on structured data stored in HDFS. To enable such queries, a secondary data store (usually a RDBMS) holds a relational schema describing the structure of the data that is stored in HDFS.

[19]https://developer.yahoo.com/hadoop/tutorial/module4.html#tolerence
[20]http://spark.apache.org/
[21]http://storm.apache.org/

Queries are compiled to run as MapReduce jobs. Hive was originally developed by Facebook [TSJ+10] to enpower ad-hoc queries by engineers and data analysts.

2.5 NoSQL

This section aims at providing a brief overview over *NoSQL* database technology. NoSQL technology is important for this thesis because the *classic* RDBMS-based way of holding data is not sufficient to achieve the level of parallelism and throughput which we would like to use for the bone simulation as an example of a complex scientific simulation.

Two different ways of exchanging data between NoSQL and classic RDBMS systems are described. Lastly, MySQL Cluster as a (perhaps surprising) example of an essentially NoSQL-based system is presented.

2.5.1 Definition and Context

A scientific simulation naturally reads input data and produces output data. It therefore requires a suitable database or at least a file system to hold data. An important consideration is how a database can scale, i. e., increase its storage capacity or query throughput given different simulation demands. Modes of scaling are

- *Vertical*: means to add more resources (i. e., CPU, disk, RAM) to a single database node.
- *Horizontal*: means to add more nodes to a (distributed) database.

Classical approaches to storing large amounts of structured data use Relational Database Management Systems (RDBMS), such as MySQL or PostgreSQL. These systems typically scale only vertically. More recently, so called *NoSQL* systems have become increasingly popular to store and query very large amounts of data. Examples of NoSQL systems include BigTable [CDG+08], Spanner [CDE+12] or Apache Dynamo. It is important to note that *NoSQL* refers less to the absence of SQL than to the absence of a relational structure [Str10]. Additionally, NoSQL systems typically possess the ability to scale horizontally.

In the context of this thesis, a RDBMS is used to persist and exchange simulation data, but data-intensive, distributed computations must be carried out in a NoSQL environment. The structure of relational databases is good for certain applications: RDBMS represent data as rows in tables that enforce a pre-defined data schema. The scheme enables very strong guarantees on how data is modified (ACID) and allows for tables to refer to each other (foreign keys). However, such systems tend to be slow at handling very large volumes of parallel accesses. This easily makes them the bottleneck during highly parallel computations, such as ours. Also, given dynamically changing loads they do not scale horizontally: a master node must holds

all data and replicate all of it to one or more slaves. Vertical scaling is easy, but inherently limited.

NoSQL databases use simpler data structures to represent their data. Many are key-value stores, similar to Distributed Hash Tables (i. e, Apache Dynamo, Apache HBase, Redis), or column-oriented (Apache Cassandra). They typically do not impose any structure on the data stored therein. The flexibility they gain from this allows them to be fully distributed and to easily scale horizontally to adapt to changing workloads. While early NoSQL databases did not, newer implementations, such as Google Spanner do provide ACID-like guarantees for some operations [CDE+12].

HDFS, and any other distributed file system, is in some sense also a (very simple) NoSQL database.

2.5.2 RDBMS-to-NoSQL bridges

In a scenario such as ours where not only SQL technologies are in use, a suitable bridge between the two worlds is needed. This means, there should be a way of importing data from an RDBMS to a NoSQL environment and also one to export it again.

Looking only at the import side, there are two possible approaches.

- *Push*: The RDBMS replicates any changes to certain tables automatically to the NoSQL system. This makes writes immediately available and allows for data streaming and incremental updates.
- *Pull*: Once data is required to be available in the NoSQL environment, all affected rows are fetched from RDBMS. This is useful to efficiently bulk load data, but makes incremental updates and data streaming hard.

For this thesis we decided to focus on Apache Sqoop, which uses the *Pull* approach.

MySQL Hadoop Applier

The MySQL Hadoop Applier is part of the MySQL project. It uses the MySQL transaction log (binary log) to instantly replicate any writes into MySQL to HDFS [MK13]. This is essentially the same mechanism through which MySQL replicates changes from master to slave nodes.

The Hadoop applier is currently (July 2014) experimental and available through MySQL Labs[22].

[22] http://dev.mysql.com/tech-resources/articles/mysql-hadoop-applier.html

Apache Sqoop

Apache Sqoop[23] uses MapReduce to pull rows from an external RDBMS into any NoSQL data store that MapReduces can directly write to i. e., HDFS or Apache HBase.

Sqoop allows a user-defined `SELECT` statement to filter input data. For example,

```
SELECT t.foo, t.bar FROM t where t.foo = 2;
```

imports only two columns from all rows matching the predicate. How the data is represented in the destination store depends on the destination store, but generally all relational structure is lost. For example when importing to HDFS, each row in the input becomes a line in a text file (other, more space-efficient import formats exist).

To parallelize the import, Sqoop first determines the total range of a user-defined *split column* in the source table. It then spawns a MapReduce job, assigning to each mapper a subset of the selected rows, using the range of the split column to partition the input. By default, the range is determined using a query like

```
SELECT MIN(t.a), MAX(t.a) FROM t
```

where `t` is the result of the `SELECT` statement that the caller invokes Sqoop with. `a` is the column whose range is to be determined (if necessary, this query can be overridden using the `--boundary-query` argument).

To achieve good import performance, the split column should be uniformly distributed within its range. However to avoid overlading the source RDBMS, Sqoop by default imports with a parallelism of 4.

MySQL Cluster

MySQL Cluster[24] is a variant of MySQL that distributes MySQL database tables across multiple data nodes. Unlike the regular MySQL server, data and indexes are held by the data nodes in main memory, allowing very low-latency access. MySQL cluster is designed as a *High Availability* (HA) system: groups of data nodes can be configured to be replicas of each other. This means a (rolling) restart of the cluster is possible without any downtime and the cluster survives failure of a fraction of its nodes.

[23] http://sqoop.apache.org/
[24] http://www.mysql.com/products/cluster/

By default, MySQL cluster still saves a log of all operations to disk such that tables can be mostly restored in the event of a full loss of the cluster (i. e., a power outage affecting the entire data center in which the cluster runs). It is, however, possible to configure the cluster to run in *Diskless* mode, in which case no logs are created. MySQL Cluster administrators can also initiate regular backups which include a full copy of all rows. Backups are useful because re-starting a MySQL Cluster instance from operation logs only is very expensive since all operations must be replayed.

The standard storage engine of MySQL cluster is called `NDB`.

While MySQL Cluster provides the full query functionality of a traditional MySQL server and is obviously a "SQL"-based RDBMS, its distributed nature also allows it to be labeled as a "NoSQL" system. Most importantly:

- It supports Horizontal Scaling since nodes can be added and removed at runtime, without affecting cluster availability.
- The distributed storage system on which the server's `NDB` storage engine is based is in fact a NoSQL system offering distributed row storage and limited query/scan functionality very similar to other key-value stores such as BigTable. This storage system is accessible independently from SQL through a lower-level C++ API[25].

A MySQL cluster consists of the following three kinds of services, all of which are typically placed on dedicated (virtual) machines:

- *Management nodes* (MGM) act as master nodes for the cluster. They keep track of all nodes participating in the cluster and coordinate backups, replication and re-balancing of tables when data nodes are added or removed from the cluster.
- *Data nodes* hold a a partition of a database table's rows in main memory.
- *SQL nodes* allow users to connect to the cluster and submit SQL queries. SQL nodes behave like a regular MySQL service. Upon receiving a query, SQL nodes compile an execution plan and execute it, utilizing the data nodes to fetch and scan rows in parallel. Data nodes are accessed using the aforementioned NDB low level API. A mechanism called AQL sometimes ships simple joins to the data nodes, in which case the SQL node only merges the results. This is advertised to yield significant speedups for joins[26]. SQL nodes operate independently of each other. If a cluster has multiple SQL nodes, an additional load balancer may be required to distribute the query load between them.

To conclude, MySQL can perhaps best be described as a blend of a classic RDBMS in terms of table structure and query expressiveness, and a NoSQL approach in terms of storing data and scaling resources. An important limitation is that while data can be accessed via multiple

[25] http://dev.mysql.com/doc/ndbapi/en/ndbapi.html
[26] http://dev.mysql.com/tech-resources/articles/mysql-cluster-7.2-ga.html

queries with very high throughput, one single query is mostly executed on a single SQL node. In comparison, MapReduce and HDFS distribute both data and computations.

3 Cluster Setup

In this chapter, Computing Clusters are defined and a brief introduction to recent advances in the field is given, followed by an overview of the infrastructure used for the bone simulation that this thesis focuses on.

3.1 Computing Clusters: Overview

The term *Computing Cluster* generally refers to a set of interconnected computers that execute distributed, compute-heavy applications. Such clusters vary greatly in size, with CPU counts ranging from tens to hundred-thousands and storage capacity up to Petabytes and more. Very large clusters usually occupy entire data centers (i. e., facilities devoted to hosting large number of computers) and thus form the backbone of companies such as Amazon, Facebook, Google or Microsoft [BCH13]. In the context of this thesis, a (very small) computing cluster is used to run the Bone simulation described in Section 2.3.

Clusters can be broadly categorized into two major classes.

- *Heterogenous*: Underlying computers span a wide range of hardware configurations and architectures. This also covers mixed CPU-GPU clusters where *Graphics Processing Units* (GPUs) provide part of the computational resources available to the cluster. Heterogenous clusters naturally occur if older and newer hardware are used together.
- *Homogenous*: All underlying computers have the same or very similar hardware and system architecture. Such clusters are therefore easier to reason about and to maintain.

Independent of this distinction, a cluster can be built either upon specialized hardware to maximize energy efficiency and performance in the cluster, or on so-called *commodity hardware* that is cheaper to buy and uses components not different than in systems that are sold to end-users.

Recent advances in datacenter design go as far as to make the analogy that a whole cluster is to be seen as a single computer in the classical sense. Hoelzle et al. write

> The software running on these systems, such as Gmail or Web search services, execute at a scale far beyond a single machine or a single rack: they run on no smaller a unit than clusters of hundreds to thousands of individual servers. Therefore, the machine, the computer, is this large cluster or aggregation of servers itself and needs to be considered as a single computing unit [BCH13].

This paradigm shift goes along with the development of toolchains that enable developers to program systems *at scale*, slowly abstracting the underlying systems away. *Virtualization* plays a key role in this: similar to how processes in a classical operating system are separated from each other through the use of virtual memory, Virtual Machines (VMs) are isolated from each other, and do not need to know about the physical hardware they run on. Similarly, Software-defined Networking (SDN) abstracts physical network topology.

If networking, storage or computing infrastructure can be provisioned, configured and deleted by use of a software service or API, it is referred to as *Infrastructure as a Service* (IaaS).

3.2 OpenStack

OpenStack[1] is an Open Source Infrastructure as a Service solution that is jointly developed by companies such as Dell, Cisco, AT&T, IBM and others, as well as by individual contributors. It provides tools to manage and provision resources (i. e., computing, storage) in a data center, in particular using virtualization technology. The infrastructure of the University of Stuttgart's Institute of Parallel and Distributed Systems, in which work for this thesis took place, is an OpenStack environment.

3.3 Apache Ambari

Apache Ambari[2] is a tool that helps with creating and operating Hadoop clusters. In our work for this thesis, Ambari is used to reliably setup Hadoop clusters.

Ambari automates the process of setting up and configuring a Hadoop cluster. It supports all major tools and services in the Apache Hadoop ecosystem, including Sqoop, Hive, Pig, Spark and Storm. Some of them (i. e., Sqoop) are client programs only, others require daemon services to be installed and configured on all or some nodes of the cluster. Some even have "Master" daemons that control cluster-wide operation (i. e., HDFS *NameNode*). A challenge is to distribute all daemons sensibly across the cluster as to avoid starvation and unevenly

[1] http://www.openstack.org/
[2] http://ambari.apache.org/

distributed loads. For this purpose, Ambari provides a default configuration that can be tailored to ones needs.

A second important use case of Ambari is to track cluster usage and health after the initial setup has been completed. Most services in the Hadoop ecosystem host simple HTML status pages on each node on which they run (usually a slightly more detailed report is available on the master node for each service respectively). However, these status pages are all independent of each other. To track the overall status and health of the cluster, one therefore has to look at a HDFS, a YARN and a MapReduce status page (or even more, depending on which services are of relevance), all spread across different hosts and with port numbers that are hard to remember. Luckily, Ambari provides a web dashboard (see Figure 3.1 and Figure 3.2) through which it makes aggregated statistics for all of the services available. Such an overview is immensely useful as otherwise a complete picture is not available.

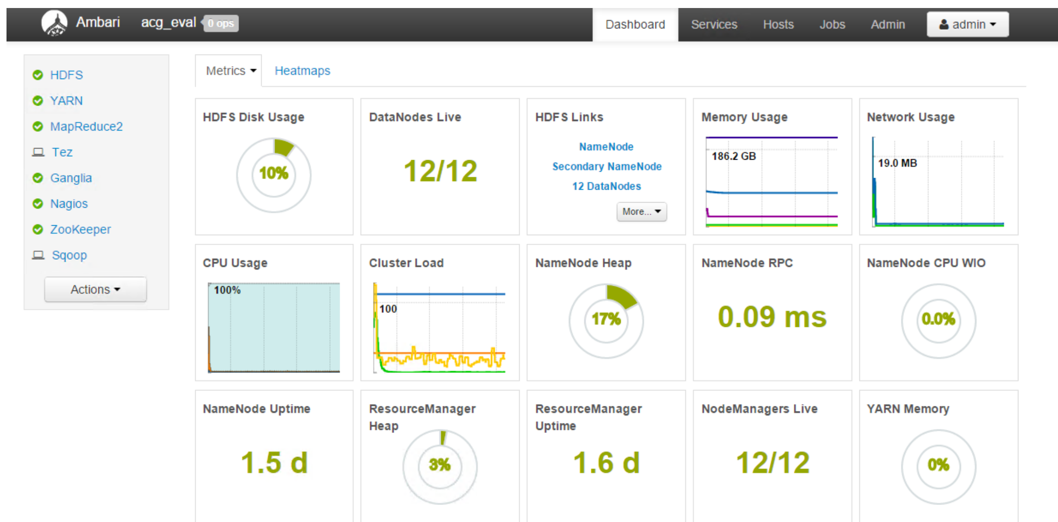


Figure 3.1: Ambari Status Dashboard (by default on port 8080)

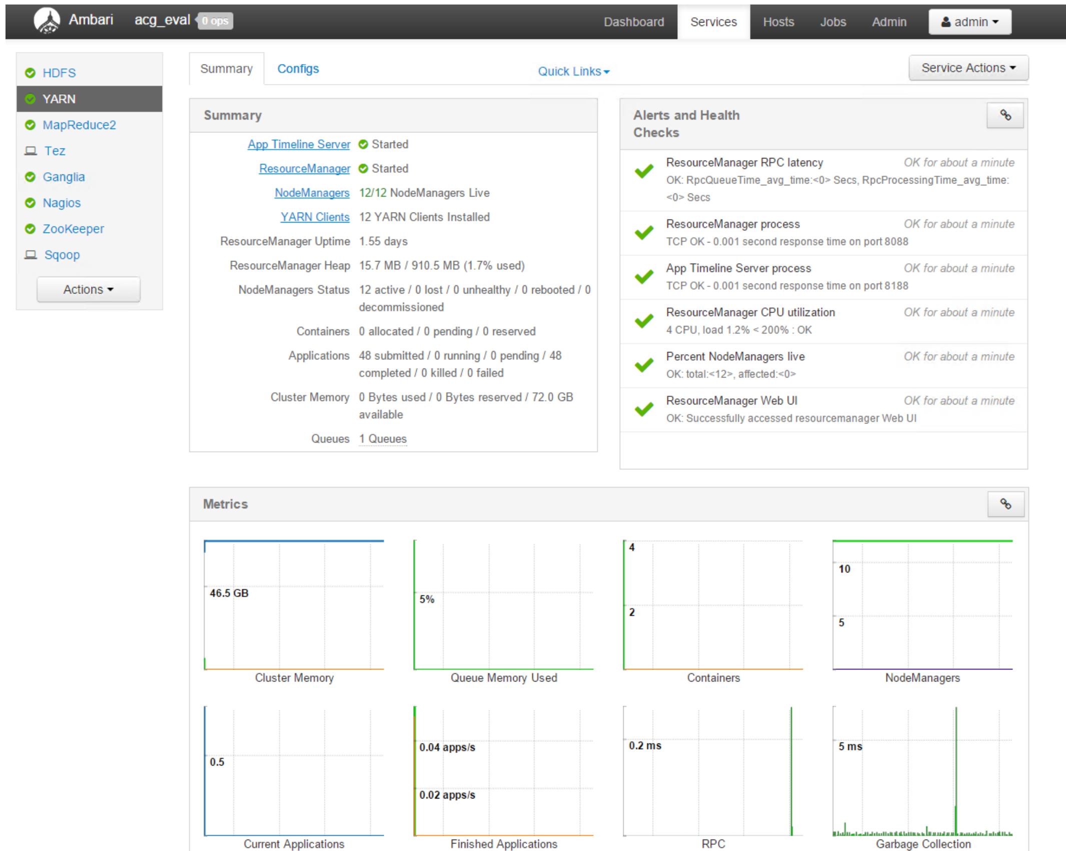


Figure 3.2: Ambari YARN Metrics Dashboard

3.4 Setup

In our setup, we create a Hadoop cluster that consists of multiple nodes. Another node, referred to as *Ambari*, hosts an Apache Ambari instance for monitoring and deployment. Additionally, a node that is referred to as *Admin*, hosts a Windows Server 2012 instance. This node is used to execute our prototypes of the simulation workflows. Lastly, a MySQL Cluster instance built from multiple nodes is set up.

A detailed description of the resources and topology available to the cluster is given in Section 6.4. This also includes a description of the physical hardware underlying the virtualized environment, which is of relevance to the evaluation.

A schematic overview of all nodes in the cluster is found in Figure 3.3, the details of those nodes are explained in the following sections.

3.4.1 Network

All nodes are within a single virtual network that is connected to the outside world via a Gateway switch. Since all nodes are within the same virtual network, they can all talk to each other. Edges in the diagram between nodes thus refer to how they actually communicate during normal cluster usage.

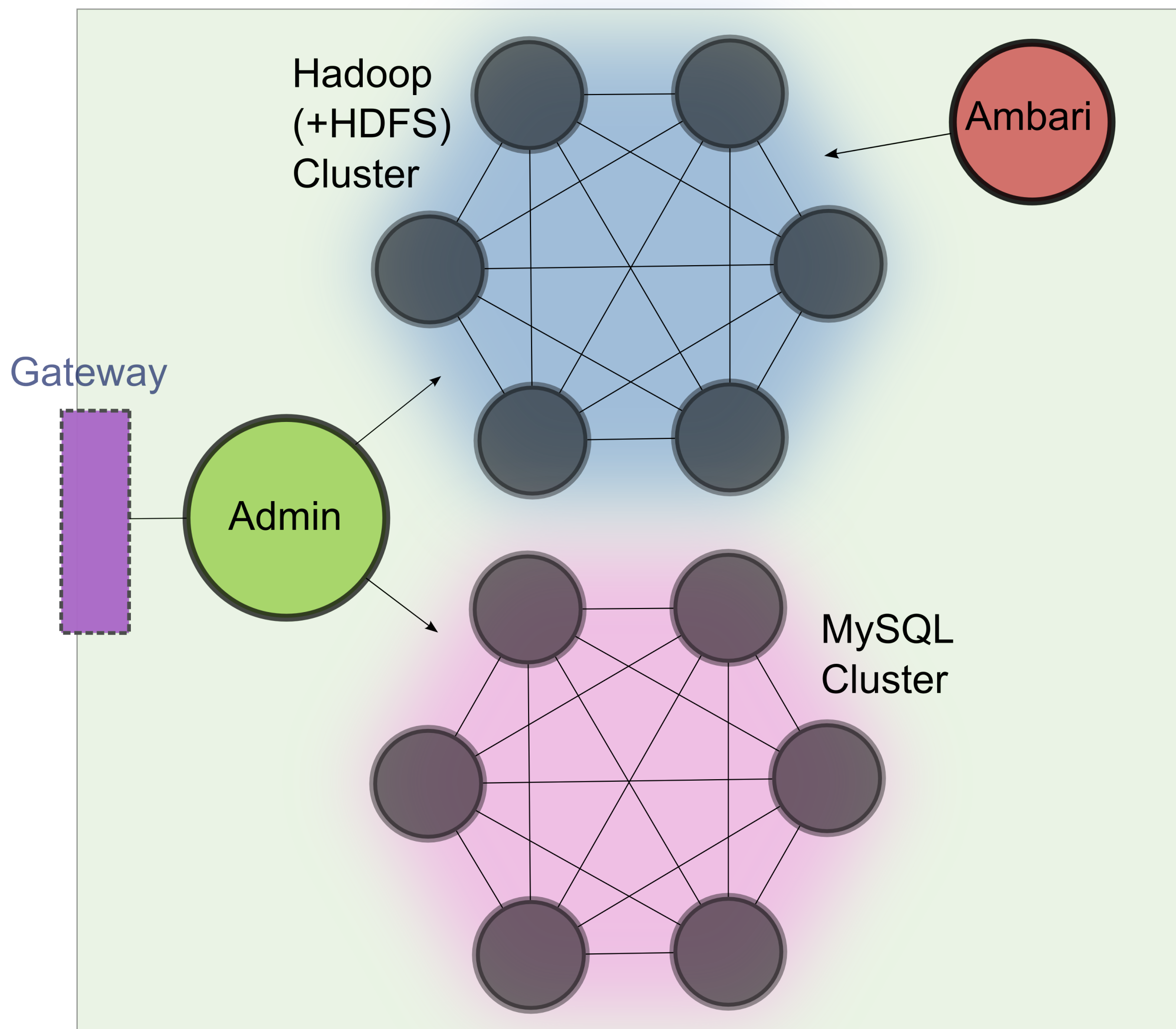


Figure 3.3: Cluster Topology

3.4.2 Admin Node

The Admin node hosts our development environment and an instance of the Apache ODE Workflow Engine[3], which is capable of executing WS-BPEL workflows. Furthermore, the node hosts an Apache Tomcat 7.0 instance[4], which supports JSP/EJB hosting (i. e., for Java-based Web Services).

The Admin node is not part of the Hadoop cluster, to start MapReduce jobs it must use SSH to issue commands on a Hadoop node.

3.4.3 Ambari Node

The Ambari node hosts an instance of Ambari. It is used for monitoring health and utilization of the Hadoop cluster.

3.4.4 Hadoop Cluster

Each of the Hadoop nodes has the same (virtualized) hardware available to it, i. e., the Hadoop cluster is a homogenous cluster as per Section 3.1.

In MapReduce/YARN terminology, each node hosts a *NodeManager*, which manages abstract *Container*s that occupy resources such as RAM, CPU cores or storage. When a MapReduce job is started, a container is launched on one of the nodes. This container hosts a *MapReduce Application Master*, which starts more containers for Map and Reduce tasks on all nodes connected to the cluster (depending on configuration and the anticipated workload) and keeps track of the overall MapReduce status.

To submit jobs (MapReduce jobs, sqoop imports/exports) to the cluster, one can use `yarn`, `hadoop` and `sqoop` binaries with parameters. Submitting jobs is equally possible from any of the Hadoop nodes, since these binaries are installed everywhere. Submitting jobs is also possible from the outside, for example from the *Admin* node.

3.4.5 Distributed Storage via HDFS

HDFS has two major components: *DataNode*s hold blocks of data and a *NameNode* maps file names to block ids and to the *DataNode*s that hold replicas of them.

[3] http://ode.apache.org/
[4] http://tomcat.apache.org/

In our setup, each of the Hadoop nodes holds a HDFS *DataNode*. One (arbitrarily picked) node also holds a *NameNode*.

Each *DataNode* stores blocks on the local disk storage of the respective node. Thus, the available HDFS storage across a cluster is less than or equal to the sum of all local disk storage available across the cluster.

3.4.6 MySQL Cluster

The MySQL Cluster instance consists of multiple data nodes, SQL nodes and one management node as described in Section 2.5.2. Each of the data nodes has the same (virtualized) hardware available to it. SQL nodes and management nodes have different needs than the data nodes and therefore run with different hardware resources.

4 Concept

In this chapter, the data flow of the current implementation of the Coupling Workflow [RSM14b] to run the combined bone simulation introduced in Section 2.3 is analyzed. In addition, common MapReduce pitfalls are explained. Then, a concept for a possible MapReduce implementation of the Octave part of the workflow is presented.

4.1 Data Flow of the Coupling Workflow

Figure 4.1 shows both data and control flow of the current Coupling Workflow based on Reimann et al. [RSM14b]. However, since the prototype and design of the workflow have evolved, the following description of the data flow is based not only on the conceptual outline in the aforementioned publication, but also on careful analysis of the existing implementation. The description is up to date as of December 2014.

At the top-level, the workflow loops over multiple *Daily Routines*. Each Daily Routine defines a schedule of *Motion Sequences* for one daily cycle, as outlined in Section 2.3.

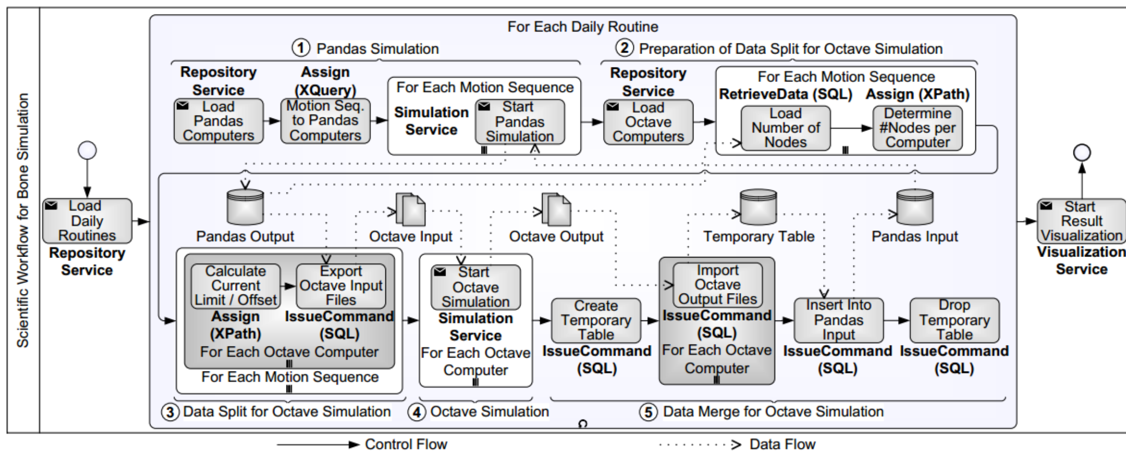


Figure 4.1: Data and control flow of the original coupling workflow [RSM14b]

The first part of each cycle is to run the PANDAS-based bio-mechnical simulation (Step 1 in Figure 4.1) for each Motion Sequence. This part is implemented through the PANDAS Workflow, which the Coupling Workflow invokes [Boh14]. The PANDAS Workflow then provisions two PANDAS instances and uses the PANDAS-WSI (Web Service Interface) to let both PANDAS instances jointly perform a Finite Element (FE) simulation for the bone. One PANDAS instance focuses on bio-mechanical simulation, whereas the second instance computes a bio-chemical simulation, which forms a rough approximation to the systems-biological simulation to be carried out using Octave). The instances use a MySQL database to exchange intermediate results.

A bone has pre-defined *FE Elements*. Each element consists of 8 predefined *Gaussian Points* for numerical integration. FE boundary conditions are given as additional input and depend on the particular Motion Sequence.

As output of a particular Motion Sequence, 22 mathematical variables for each Gaussian Point and timestep are written to a MySQL table in a key-value structure that allows the two PANDAS instances to write their results independent of each other. The schema of this output table is shown in Listing 4.1.

Listing 4.1 SQL schema of the PANDAS output table

```
CREATE TABLE pandas_output(
  -- Unique ID of this Motion Sequence
  motionid INT,
  -- ID of the Timestep
  timestep INT,
  -- ID of the FE Element Point
  elementid INT,
  -- ID of the Gaussian Point
  gaussid INT,
  -- Name of the mathematical variable
  name VARCHAR(10),

  PRIMARY KEY(motionid, timestep, elementid, gaussid, name)

  -- Output variable value
  value DOUBLE
);
```

Of all variables outputted for a single `(motionid, timestep, elementid, gaussid)` key tuple, only the variables named `NS0`, `SIG_V` and `CNUF` are later used. Furthermore, to simplify the load on the Octave simulation, the *average* (mean) value of these variables across all timesteps (`timestep` column) is calculated and used instead (for each Motion Sequence).

In a SQL formulation this corresponds to Listing 4.2, where `CURMOTIONSEQ` indicates the current Motion Sequence ID (`motionid`). Note that the query is unoptimized.

Listing 4.2 SQL query to calculate aggregate variable values from `pandas_output`

```
SET @CURMOTIONSEQ = /* <motion sequence to be processed> */;

SELECT A.elementid, A.gaussid, NSTS_avg, SIG_V_avg , CNUF_avg
FROM

( SELECT elementid, gaussid, AVG(value) as NSTS_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'NSTS'
GROUP BY elementid, gaussid
) AS A,

( SELECT elementid, gaussid, AVG(value) as SIG_V_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'SIG_V'
GROUP BY elementid, gaussid
) AS B,

( SELECT elementid, gaussid, AVG(value) as CNUF_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'CNUF'
GROUP BY elementid, gaussid
) AS C

WHERE A.elementid = B.elementid AND A.gaussid = B.gaussid
  AND B.elementid = C.elementid AND B.gaussid = C.gaussid
```

For each Motion Sequence, `(elementid, gaussid, NS0_avg, SIG_V_avg, CNUF_avg)` tuples are exported to separate CSV files (Step 3 in Figure 4.1). The Coupling Workflow then invokes the Octave Workflow which lets Octave perform the systems-biological bone simulation for each `(elementid, gaussid)` tuple during the day (Step 4 in Figure 4.1) [Boh14].

The schedule for an entire day of activity is put together according to the current Daily Routine (see again Figure 2.3). For example if the available Motion Sequences are *Sleeping, Walking, Working* and *Watching TV* then a possible schedule for a 24 hour day is

```
6  hours Sleeping
1  hours Walking
11 hours Working
1  hours Walking
4  hours Watching TV
1  hours Sleeping
```

Accordingly, the Octave simulation processes each `(elementid, gaussid)` tuple by fetching the variable values for this tuple from the individual CSV files in the order of the schedule (this

means, some variable values are used multiple times as a Motion Sequence can occur multiple times during a day; the length of an activity also affects the simulation). For each (elementid, gaussid) tuple, Octave produces two output variables, NSTS and HCF.

This result is written into another MySQL table, the scheme of which is found in Listing 4.3 (This corresponds to the *Temporary Table* in Step 5 of Figure 4.1).

Listing 4.3 SQL schema of the Octave output table

```
CREATE TABLE octave_output(
  -- ID of the FE Element Point
  elementid INT,
  -- ID of the Gaussian Point
  gaussid INT,

  PRIMARY KEY(elementid, gaussid)

  -- Output variables
  NSTS DOUBLE,
  HCF DOUBLE,
);
```

The NSTS and HCF variables serve as input constants to PANDAS for the next Daily Routine. For this, they are re-imported into the data store that holds inputs to PANDAS.

This completes the data flow for one cycle of the Coupling Workflow.

4.2 Data Size

To properly analyze the workflow, it is relevant to have an estimate of the magnitudes of the data involved. With respect to the pandas_output table from Listing 4.1, the range of values for each column of the table is shown below.

motionid	1-100
elementid	1000-10000
gaussid	8
timestep	10-1000
name	22

This means pandas_output contains between 1.7 million and 176 billion rows. This is 13.42 MiB to 1.28 TiB of data in the value column.

A maximum estimate for the entire table's storage size is about 3.52 TiB[1] if the following assumptions are met:

1. Instead of the illustratory `INT`, the smallest possible integer types based on the estimate are used for the `id` columns, i. e., `TINYINT` for `motionid`, `SMALLINT` for `elementid`, `TINYINT` for `gaussid`, `SMALLINT` for `timestep`.
2. No alignment or padding occurs.
3. (Most) variable `names` have length 5.
4. There is no further storage optimization (e. g, compression)

This amount could be further decreased by hard-coding output variables as columns instead of using the `name` column, which however is the preferred output format for PANDAS.

It should also be noted that this is still only for the simulation of exactly one bone.

4.3 Appropriate Use of MapReduce

To be able to efficiently formulate parts of the given simulation using the MapReduce paradigm, it is important to first understand what makes MapReduce *fast*. Some computations simply do not map well to MapReduce, other do so trivially and others can be re-formulated to work.

The goal of this section is to identify reasons why a specific part of the workflow does not map well to the MapReduce paradigm and should be parallelized using other means, if at all.

4.3.1 Under-utilization

In both the mapper and reducer stage, input data for the relevant stage is partitioned across all available worker slots.

The number of worker slots is usually configured to be the number of cores in the cluster. For efficiency, Hadoop defines a minimum amount of input data that a worker slot must process during the map stage (referred to as *input split* size). The heuristic makes sense: the cost of setting up a distributed job is very high, it involves the cost of scheduling as welll as copying mapper and reducer binaries. However, if the input data is less than the input split size times the number of available workers, some worker slots are idle during the map stage. This can be a problem if the input size is very small even though the computation time per input key is high. The problem can be mitigated by tweaking the Hadoop configuration.

[1] http://dev.mysql.com/doc/refman/5.0/en/storage-requirements.html

For the reduce stage all the *unique* keys emitted by the map stage are partitioned across all available worker slots. Thus, if the map stage emits less *unique* keys than there are worker slots, some worker slots will be idle during reducing! This can be hard to overcome and is a sign that the task in question does not exhibit sufficient parallelism.

4.3.2 Bad partitioning

The partition of the inputs across all available worker slots is closely related to hashing keys to buckets in a hash table: ideally, each worker slot would be assigned a share of the input that requires equal computation effort to process, but in practice the distribution of said effort is not known a priori. This occurs frequently at the reduce stage when a few input keys have significantly more values than others and therefore take longer to reduce than others. To make this distribution more uniform, one can manually tweak Hadoop's `Partitioner` function, which controls the mapping from keys to worker slots (i. e., the hash function in analogy) as to ensure that the "slow" keys are spread out uniformly across all worker slots. A good example where this occurs is the classic `WordCount` example uncannily often found in most MapReduce introductions: since the distribution of words in most natural languages approximately follow Zipf's Law[2], few words account for a large proportion of all words in a source text. Luckily, there are enough words, so most choices of `Partitioners` should do reasonably well at distributing the work along reducers. It is likely though that some of those reducers take much longer than the average.

4.3.3 Large amount of intermediate storage

If the mapper stage emits more or larger records than it reads, a large amount of intermediate data is produced on each mapper, where it is periodically sorted and eventually queued up on disk (*spilled* in Hadoop terminology). As part of the Shuffling stage, this potentially requires a costly on-disk merge step on *both* mapper and reducer worker slots[3].

This easily happens if complex SQL-like Extraction/Transform/Load (ETL) statements are "converted" to MapReduce. As an example, consider you have as input a set of email address books, each of which lists multiple email addresses. An additional input is a list of pairs of email addresses and the desired output is a subset of this list containing only the email address pairs that co-occur in at least one address book. A possible MapReduce implementation could use one mapper function to emit ((address, address), address-book-id) pairs for each pair of addresses in the input address books and a second mapper function to emit ((address, address), NIL) pairs for each pair of email addresses in the input email address pair list (it is possible to

[2]http://en.wikipedia.org/wiki/Zipf's_law
[3]http://de.slideshare.net/cloudera/mr-perf

specify different mapper functions for different parts of the input). The reducer function then could check if, for each pair of email addresses, at least one address-book-id and exactly one NIL value is found. Clearly, the intermediate output required by this implementation would grow quadratically.

Sometimes such problems can be avoided by using a sequence of MapReduces instead of a single MapReduce. In this example, a first MapReduce job could generate an inverted index from email addresses to address books that contain them. A second MapReduce could then intersect the address book lists for all email addresses. The two-stage solution requires only $\Theta(n)$ intermediate storage. The differences in intermediate storage does not even have to be asymptotical in order to be problematic: even a 5-fold consumption of intermediate storage with respect to input size can easily become a performance problem, not to mention the amount of spare HDFS capacity it requires.

4.3.4 Use of external Web Services and Files

In a service-oriented Architecture (SOA), even simple computations often depend on a multitude of external Web Services. With MapReduce, this can be problematic: any services that are being accessed from within a mapper or a reducer need to be highly replicated as to cope with the enormous query rate that a Hadoop cluster of only few nodes can generate. Such behaviour also greatly increases the complexity of the entire system, for example network bandwidth and congestion can become a concern. Similarly, if jobs fetch additional, shared data from HDFS upon starting work on an input partition, the read capacity of the file system is quickly exhausted. Increasing HDFS replication can help, but does not solve it altogether.

As a rule of thumb, MapReduce jobs should not require any inputs besides their primary inputs and they should not talk to Web Services in case the implementation of the Web Services is not at least as scalable as the Hadoop cluster.

4.4 Parallelization of the Workflow using MapReduce

To make any given part of the coupled simulation workflow a viable candidate for a MapReduce-based implementation, two requirements must be met:

1. Its data flow makes for an efficient mapping to MapReduce as per Section 4.3.
2. All its dependencies can easily be made available on the Hadoop cluster. For example, running an Octave script requires a local Octave installation on each node in the cluster. In this thesis such dependencies are considered a prerequisite and their installation (Platform Provisioning) is not covered. However, dependencies that are expectedly hard to fullfil do not make good candidates for a MapReduce-based implementation.

The following is an analysis of the data flow in the workflow with respect to these requirements, in the order of the steps in Figure 4.1 (step number in parentheses).

1. At a very top-level: multiple bones in a skeleton could be simulated independently. This is outside the scope of the Coupling Workflow though, which performs the simulation for one bone only. Therefore, this is not considered part of this work. However, a possible area of further inquiry is to run a “global” MapReduce that runs the entire simulation for each bone in a mapper, and uses the reduce step to generate results such as average bone density across the entire skeleton.

2. (Step 1) The bio-mechanical PANDAS simulation runs data-independent for each Motion Sequence. Parallelizing it could yield speedup linear in the number of Motion Sequences, but requires both PANDAS-WSI and PANDAS itself to be deployed across the cluster as part of Platform Provisioning, along with all static input data required for the simulation (i. e., Gaussian Points). The way how two PANDAS instances interact during the bio-mechnical simulation of one Motion Sequence is complex and heavily depends on MySQL. Several ways to avoid this can be imagined: PANDAS could be rewritten to use a column-oriented, key-value store that scales horizontally, such as Apache Cassandra (if its eventual consistency can be coped with), or HDFS directly. MySQL instances could be deployed locally on each node of the Hadoop cluster, and final results merged into a central instance afterwards. An alternative is to use a sufficiently powerful MySQL cluster instance. All of these approaches require a major re-engineering of the PANDAS simulation and have complex platform demands. Even then, the result does not map well to MapReduce for the reasons given in Section 4.3.4.

   Parallelizing an instance of the PANDAS simulation in itself is not feasible because the simulation involves solving a complex field equation with complex data dependencies.

3. (Step 3) *Averaging PANDAS output over all timesteps.* This is done independently for each `(motionid, elementid, gaussid, name)` primary key tuple. A mapper function can collect all values to be averaged for each primary key tuple. A reducer function then calculates the mean of all collected values. To maximize Reduce worker utilization, the number of timesteps should be the same for all primary keys. Different Motion Sequences can have different length and thus different number of timesteps though, causing potentially bad partitioning (see Section 4.3.2). However, for a a large number of Motion Sequences this will be negligible, and for a small numbers of Motion Sequences it is the largest Motion Sequence that dictates the worst case runtime.

   A disadvantage is that plain MapReduce for this task involves producing a full copy of all data to be averaged as intermediate mapper output (see Section 4.3.3). However, this can be alleviated by using a `Combiner` that partially reduces the data at each mapper worker.

4. (Step 4) *Running the Octave simulation.* This part has perhaps the highest potential for speedup: The number of `(motionid, elementid, gaussid)` tuples to be processed is significantly larger than the number of available MapReduce worker slots. The process of dropping the `motionid` and generating a sequence of tuples in the order of the activity schedule corresponding to the Daily Routine maps directly to a single MapReduce step: the mapper emits tuples along with the time of day (index of hour) at which they occur as an extra key (multiple copies of each tuple if they occur multiple times during a day). The reducer then sorts the sequence of values by time of day and proceeds to reduce it in the correct order. Automatic sorting by such a secondary key can also be achieved using a special sorting trick referred to as *Secondary Sort*[4], which involves using custom comparer and partitioner implementations to Hadoop. Also, since the Octave simulation consists of pure arithmetic without branches, its runtime depends only on the input length, which would be constant for each `(elementid, gaussid)` pair.
5. (Step 5) *Data merge.* In this step, data is merged and written back into the table holding future inputs. Since the destination is a single relational database and the task involves no computation, the MapReduce paradigm is not applicable here.

From this the following execution plan for the entire Coupling Workflow follows.

1. Run the PANDAS simulation as usual, write results into the `pandas_output` table (Listing 4.1).
2. Import relevant parts of the `pandas_output` table into HDFS.
3. **MapReduce1**: A first MapReduce job reduces the `timestep` column and calculates average values.
4. **MapReduce2**: A second MapReduce job reduces the `motionid` column as per the Daily Routine and calculates outputs using Octave.
5. Export the result back into a MySQL table `octave_output` (Listing 4.3).

The execution plan is summarized in Figure 4.2, which shows both data and control flow. The first stage, running the PANDAS simulation, is omitted under the assumption that the output of PANDAS is pre-existent in the `pandas_output` table.

4.5 MapReduce as a Service

One goal of this thesis is to explore how MapReduce computations can seamlessly integrate into simulation workflows. Workflow authors would like to focus on the actual logic and

[4]http://www.quora.com/What-is-secondary-sort-in-Hadoop-and-how-does-it-work

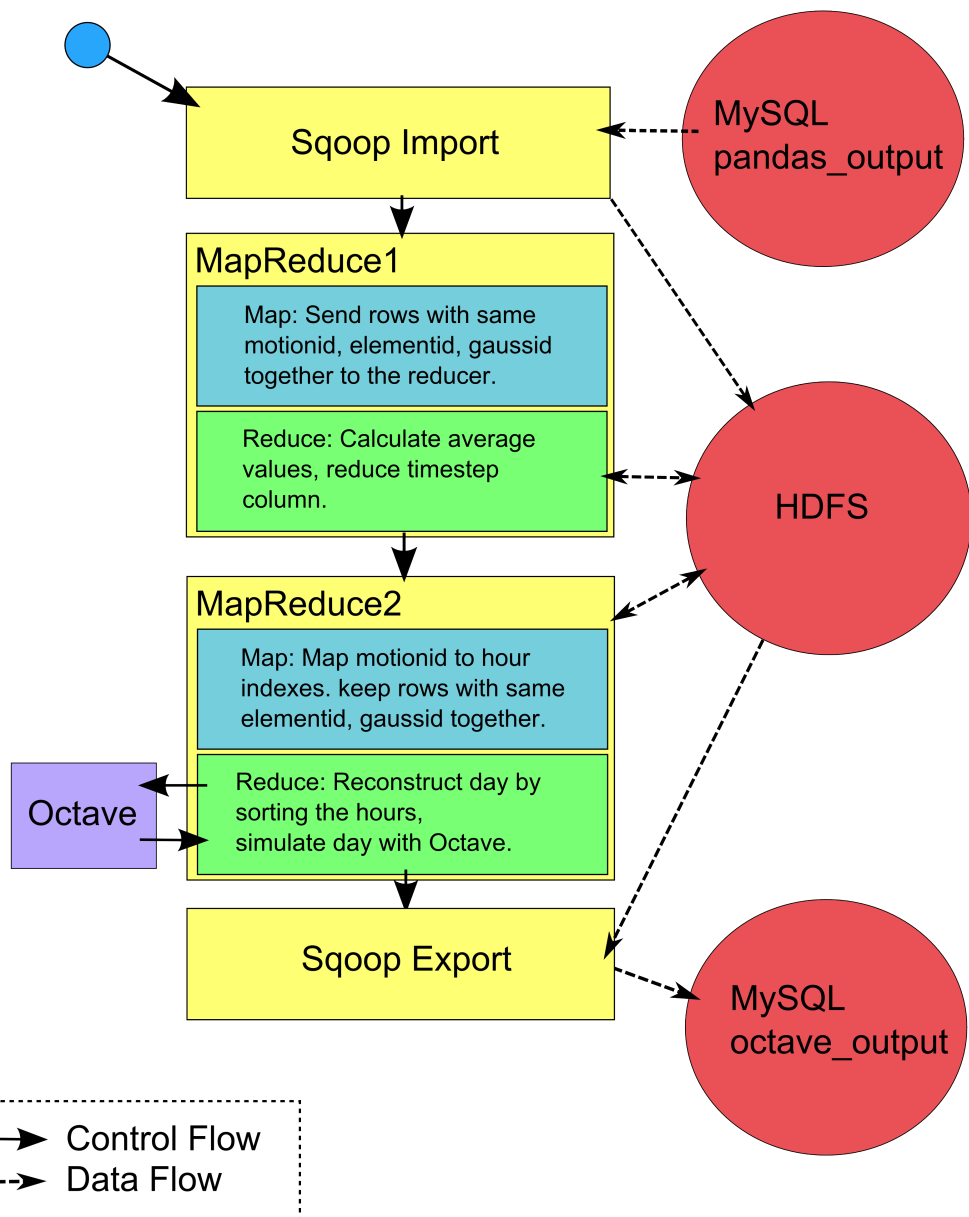


Figure 4.2: Data and Control Flow of the MapReduce prototype

not be bothered with too much implementation detail. The MapReduce version of the bone simulation, as outlined in the previous section, should be simple. Thus the question is how MapReduce jobs and HDFS imports and exports are executed from within a workflow.

The SIMPL framework provides the *IssueCommand* executable data management operation to invoke arbitrary shell commands as part of a workflow (see Section 2.2.2). This would be a very straightforward way to run our MapReduce jobs if the necessary Hadoop binaries were available on the node that executes the Simulation Workflow. In our setup though, and from a more general perspective, it cannot be assumed that this is the case. Running remote MapReduces is still possible with a shell command, but requires `ssh` and potentially `scp` to a remote Hadoop node (however, the SIMPL framework already offers the functionality to issue shell commands over SSH interfaces, but it still requires workflow authors to specify the relevant shell command). To hide these complications from workflow authors, the concept for this thesis is to wrap basic MapReduce functionality in a Web Service that seamlessly integrates into workflows. The service can also provide HDFS import and export functionality.

Furthermore, the whole process of writing mapper and reducer functions as Java classes and packaging them up in a JAR to deploy them across the Hadoop cluster is tedious: many real-world mapper and reducer functions are surprisingly simple so the ratio between boilerplate code and actual logic is oftentimes high. For our MapReduce service, the concept is to use Hadoop Streaming mode. This allows workflow authors to formulate their mapper and reducer functions in a lightweight scripting language that does not require compilation (i. e, Python).

5 Implementation

This chapter describes the implementation of a prototype for the parallelization concept outlined in the previous chapter. First *MapReduce-WSI* is described, which enables running MapReduce jobs with very little effort. Then, the implementation of the *MapReduce prototype*, which builds on MapReduce-WSI is presented. Lastly, the *baseline prototype*, which represents a manual approach to parallelizing the Octave simulation to multiple nodes, is discussed. In Chapter 6, the MapReduce prototype will be compared against the baseline prototype.

5.1 MapReduce-WSI

MapReduce-WSI implements a simple Web Service frontend to MapReduce on a Hadoop cluster. The Web Service allows users to easily import data from a JDBC-based RDBMS into HDFS, run custom MapReduce jobs and eventually export the results back into the RDBMS. Written as part of this thesis, it has been released as Open Source and its source code is freely available on Github[1]. MapReduce-WSI builds on top of the JAX-WS API that is part of Java SE 1.6. There are no dependencies other than Hadoop and the SSHXCUTE[2] framework, which however is bundled in the source code repository.

In our environment (see Section 3.4), an instance of MapReduce-WSI runs on the *Admin* node.

The Web Service has been kept stateless as to avoid requiring client support for WS-Adressing or HTTP Sessions. However, a logical state is introduced by the so-called `scope-id`, which identifies an isolated environment for individual users.

The typical workflow when using MapReduce-WSI is as follows:

1. First call the `createScope` API, which allocates a new scope-id and returns it to the client. All further service invocations receive this scope-id as their first parameter in order to properly identify the isolated environment to be used for the invocation.

2. Import zero or more tables from a RDBMS into HDFS using the `importIntoHDFS` API.

[1]https://github.com/acgessler/mapreduce-wsi
[2]https://code.google.com/p/sshxcute/

3. Run zero or more MapReduce jobs on the imported data. There are two APIs for this: `runMapReduce` exposes the full Hadoop MapReduce functionality but requires more work to use (since mapper and reducer have to be compiled into a JAR archive) whereas `runStreamingMapReduce` offers a simple way of specifying mappers and reducers. Both options are further explained in the next sections.
4. Export the output from HDFS to a RDBMS using the `exportFromHDFS` API.
5. Use `deleteScope` to permanently delete all resources on the Hadoop cluster that were associated with the scope-id.

The full WSDL document for MapReduce-WSI can be found in Appendix A.

5.1.1 Isolation

Being a WebService, MapReduce-WSI can be used by multiple clients at the same time and it is critical to isolate them from each other. To achieve a basic level of isolation, we use the scope-id to identify an unique environment. For each scope-id, a new HDFS folder is maintained. In the current implementation, scope-ids are drawn from a random number generator (RNG), which ensures uniqueness with sufficient probability. Therefore there is no database that stores active scope-ids and when allocating a scope-id, the corresponding folder is created. All HDFS file paths used by the user with the scope are rewritten to use this folder as root. This happens completely transparent to the user and avoids accidental data conflicts. However, the system is (at least in the current prototype) not secure: there is no path normalization and the user-defined JAR archives that `runMapReduce` deploys to the cluster run with full HDFS privileges and can theoretically access other user's data.

It is important to understand that, while different scope-ids enjoy said weak file system isolation, the computational resources in the cluster are limited and simultaneously running jobs do affect each other's runtime. While YARN's scheduling mitigates some of those effects, care must be taken not to over-load the cluster, in particular with respect to filesystem use.

5.1.2 runMapReduce

`runMapReduce` deploys a JAR archive from the local machine onto the cluster and executes it using the command `yarn jar`. The JAR archive must contain the mapper and reducer classes as well as a `main()` method that contains the standard MapReduceV2 bootstrap code, using whichever configuration the client wants. An example can be found in the repository[3].

[3] https://github.com/acgessler/mapreduce-wsi/tree/master/test

The API allows forwarding arbitrary arguments to the remote processes. Additionally, as very first argument the JAR is provided with the HDFS root folder of the current scope-id (such that it can locate its inputs).

5.1.3 runStreamingMapReduce

`runStreamingMapReduce` uses Streaming Mode to run a MapReduce job. The source code for a mapper and reducer script are specified as strings. The scripts can be formulated in any language, provided the interpreter for said language is available on all nodes in the cluster. In addition, a valid UNIX Shebang[4] must be present to make the scripts shell-executable. For a valid example, see again Listing 2.1 and Listing 2.2.

Since the source code for the scripts is transmitted as a parameter to the service, it is not recommended to use this method for long scripts. Another shortcoming is that scripts that references to other files (e. g., Python or Octave module imports) are not supported.

HDFS data source and destination are specified directly in the API call. The user has no further control over the MapReduce configuration (i. e., parallelism or sharding): if such level of control is needed, `runMapReduce` should be the API of choice.

5.1.4 importIntoHDFS

`importIntoHDFS` uses Apache Sqoop (see Section 2.5.2) for bulk importing. The user specifies a JDBC database source with credentials, a HDFS destination location, a `SELECT` statement (user query) and split column as described.

The implementation[5] avoids a current shortcoming of Sqoop: The way how Sqoop derives the default boundary query on the split column fails if the split column itself is not selected by the user query. In the context of this thesis, this would cause trouble multiple times during the simulation.

For example, in a table with three columns named *a*, *b* and *c*, a possible user query could be

```
SELECT b, c
  FROM X
  WHERE Y
;
```

[4]http://en.wikipedia.org/wiki/Shebang_(Unix)
[5]https://github.com/acgessler/mapreduce-wsi/blob/master/src/de/uni_stuttgart/ipvs_as/MapReduceWSIImpl.java#L221

However, if in this case a (which is not selected!) is specified as split column, then the incorrect boundary query generated by Sqoop is

```
SELECT MIN(t.a), MAX(t.a) FROM (SELECT b, c
  FROM X
  WHERE Y
) t;
```

The implementation of `importIntoHDFS`, however, correctly detects this case and emits a boundary query that rewrites the original `SELECT` statement to include the split column a while leaving the query used for selection untouched:

```
SELECT MIN(t.a), MAX(t.a) FROM (SELECT a, b, c
  FROM X
  WHERE Y
) t;
```

5.1.5 exportToRDBMS

The `exportToRDBMS` API is the complement to `importIntoHDFS`. It also uses Apache Sqoop and exports tuples from a HDFS path to a given relational database table, mapping each source line to a row and each element within a line to a column (left to right). The destination is again specified as a JDBC URI with credentials. Unlike `importIntoHDFS` no filtering on the data is supported and the destination table's key constraints must allow for the source data to be inserted individually.

For example, if the HDFS source path contained the following tuples:

```
0 24215
1 991
0 241
```

And the destination table schema is:

```
CREATE TABLE t(
  a INT,
  b INT
);
```

Then `exportToRDBMS` is equivalent to the following SQL statement

```
INSERT INTO t(a, b) VALUES (0, 24215), (1, 991), (0, 241);
```

However, exporting would fail if t puts a primary key constraint on a, because the values inserted into this column are not unique.

5.2 MapReduce Prototype

This section describes the implementation of the MapReduce prototype, which is later evaluated against a baseline prototype (see Section 5.3). The prototype is implemented in Java and only covers the Data aggregation Octave part of the Coupling Workflow. PANDAS-WSI is not invoked because the concept does not plan parallelizing it, so it is seen as a constant. The prototype reads its input from the `pandas_output` table as shown in Listing 4.1. This table can either be populated by manually running PANDAS, or by artificially generating test data.

The prototype closely follows the concept outlined in Chapter 4, in particular the control and data flow found in Figure 4.2.

5.2.1 Import into HDFS

Only a subset of the PANDAS output table needs to be imported into HDFS. Since the Octave simulation only requires 3 out of the 22 `double` mathematical variables that the PANDAS simulation calculates, most of the input can be discarded. The `SELECT` statement that is passed to `importIntoHDFS` becomes simply

Listing 5.1 Sqoop import statement

```sql
SELECT motionid, elementid, gaussid, timestep, name, value
  FROM pandas_output
  WHERE (name = 'NS0' OR name = 'SIG_V' OR name = 'CNUF')
```

As split column (see again section 2.5.2), the `elementid` column is used: it contains integer IDs that are uniformly distributed and has enough distinct values (see Section 4.2) to allow partitioning in as many approximately equal parts as mapper slots used by Sqoop.

This produces multiple HDFS text files that contain 6-tuples on each line. The number of individual files depends on the number of workers that Sqoop uses to import the data (4 by default). This is not of relevance to further steps though: if the size of the input data exceeds the *input split size* (see Section 4.3), MapReduce workers will be assigned to subsets of an input file.

5.2.2 Averaging timesteps

This stage is relatively straightforward: given `(motionid, elementid, gaussid, timestep, name, value)` 6-tuples, compute for each `(motionid, elementid, gaussid, name)` 4-tuple the average (mean) of the `value` column across all time steps. The calculation happens only for the 3 variables of interest. In the MapReduce formulation, the mapper function discards the time step and emits all values for which to calculate the mean under the same key. The reducer function performs the actual calculation and emits the mean values of the 3 variables as output columns in a single row.

A problem with a naive calculation of the mean (i. e., sum all values up and divide by the number of values) is that the sum is prone to overflows. To avoid, we utilize an algorithm for calculating the mean iteratively. This avoids any overflows, and is numerically stable [Knu97].

As an additional optimization to save intermediate space and string processing time, the names of the variables processed are compressed into numeric indexes.

The full implementation of the Python scripts of the mapper and reducer functions for this stage can be found in Listing 5.2 and 5.3.

A shortcoming is that the current implementation does not use a combiner function to partially reduce data at each mapper slot because MapReduce-WSI does not support combiner functions (yet). This could significantly decrease the amount of intermediate storage required.

Listing 5.2 AverageTimesteps mapper script

```
#!/usr/bin/env python

# Input:
# (motionid, elementid, gaussid, timestep, name, value)
#
# Output:
# (motionid_elementid_gaussid, nameid, value)

import sys

# This table maps variable names to a simple numeric index
namemap = {
  'NS0' : 0,
  'SIG_V' : 1,
  'CNUF' : 2
}

for line in sys.stdin:
  tuple = line.strip().split(',')
  assert len(tuple) == 6

  # Replace the variable name with a numeric index
  # to reduce the size of intermediate HDFS storage.
  nameid = namemap[tuple[4]]

  # Emit the concatenation of the primary key 3-tuple as key
  # using _ to separate the parts. This has the effect that
  # values with the same primary key are mapped to the
  # same Reducer shard while we can still reconstruct the
  # original key later.
  key = '_'.join(tuple[:3])
  print '%s\t%i\t%s' % (key, nameid, tuple[5])
```

Listing 5.3 AverageTimesteps reducer script

```
#!/usr/bin/env python

# Input:
# (motionid_elementid_gaussid, nameid, value)
#
# Output:
# (motionid, elementid, gaussid, mean(NS0), mean(SIG_V), mean(CNUF))

import sys
import itertools

mean = [0] * 3

lines = (line.split() for line in sys.stdin)
for key, values in itertools.groupby(lines, lambda x : x[0]):

  # Iterative mean calculation
  # See The Art of Computer Programming, Volume 2, section 4.2.2 [Knu97]

  mean[0] = mean[1] = mean[2] = 0.0
  for index, value in enumerate(values):
    assert len(value) == 3

    nameid = int(value[1])
    mean[nameid] += (float(value[2]) - mean[nameid]) / (index + 1)

  # Emit the average values with the original (split) key
  print '%s\t%s\t%s\t%f\t%f\t%f' % tuple(key.split('_') + mean)
```

The result is written to HDFS and directly re-used for the next stage.

5.2.3 Octave Parallelization

The second Mapreduce stage also involves a separate mapper and a reducer written in Python, however the reducer script is paired with an Octave instance to perform the actual calculation. Given the `(motionid, elementid, gaussid, mean(NS0), mean(SIG_V), mean(CNUF))` 6-tuples generated by the previous Mapreduce stage, the Python mapper and reducer together generate for each unique pair of (`elementid`, `gaussid`) a sequence of all variable values (means of `NS0`, `SIG_V` and `CNUF`) ordered over the course of a Daily Routine of 24 hours. The algorithm is that the mapper emits tuples together with an additional index that represents the hour. If a Motion Sequence is active during multiple hours of the day, the same values are emitted multiple times using different hour indexes. The reducer then sees all the values for a (`elementid`, `gaussid`) key tuple and sorts them by the hour indexes, thus achieving a reconstruction of the Daily Routine with respect to the variables under simulation.

In the prototype, the mapping from hours of day to the corresponding Motion Sequence is directly embedded into the mapper script. The resulting sequence is then forwarded to an Octave instance which runs in parallel to other Octave instances and calculates the final result of the simulation with respect to the (`elementid`, `gaussid`) pair and the Daily Routine. This coupling step also happens as part of the reducer script. The Python reducer script collects the result from Octave and writes them to HDFS.

Note that every reducer script instance starts its own instance of Octave. Also, the entire reducer could be implemented in Octave alone, yet Python makes it easier to formulate the sorting logic wheras Octave's strength is to perform the actual calculation.

The Octave script used with the prototype is not the original simulation because it requires to read several sources of configuration to get simulation constants. The simulation script for this thesis was designed to mimick the computational effort of the original as closely as possible and uses Octave's `lsode` library function[6] to solve an Ordinary Differential Equation (ODE) defined on a function of the input vectors.

The implementation of the Python scripts of the mapper and reducer functions for performing the Octave simulation is found in Listing 5.4 and 5.5. The reducer is partly given in pseudo-code, the details of the communication with Octave using Python's `subprocess` module and a two-way pipe to redirect `stdin` and `stdout` of the Octave process are omitted for brevity.

[6]https://www.gnu.org/software/octave/doc/interpreter/Ordinary-Differential-Equations.html

Listing 5.4 OctaveCalc mapper script

```
#!/usr/bin/env python

# Input:
# (motionid, elementid, gaussid, mean(NS0), mean(SIG_V), mean(CNUF))
#
# Output:
# (elementid_gaussid, hour, mean(NS0), mean(SIG_V), mean(CNUF))

import sys

# This table contains 24 entries corresponding to 24 hours in a day.
# For each hour, it specifies the index of the motion sequence
# that describes the activity during the hour.
hour_motionids = [
  0,0,0,1,1,1,0,1,
  0,0,0,2,2,1,0,1,
  0,0,0,1,1,1,0,1
]

for line in sys.stdin:
  tuple = line.strip().split()
  assert len(tuple) == 6

  motionid = int(tuple[0])

  # Again, emit a composite key to enable reduction on pairs of
  # elementid, gaussid. Each time the motionid is used during
  # the day, a new entry is emitted using the number of the
  # hour as additional tuple element.
  key = tuple[1] + '_' + tuple[2]
  for hour, hour_motionid in enumerate(hour_motionids):
    if hour_motionid != motionid:
      continue
    print '%s\t%i\t%s\t%s\t%s' % (key, hour, tuple[3], tuple[4], tuple[5])
```

Listing 5.5 OctaveCalc reducer script

```
#!/usr/bin/env python

# Input:
# (elementid_gaussid, hour, mean(NS0), mean(SIG_V), mean(CNUF))
#
# Output:
# (elementid, gaussid, NSTS, HCF)

import sys
import itertools

<START OCTAVE INSTANCE>

# Incoming elements are grouped by elementid_gaussid, but
# not ordered with respect to the hour column. Thus, sort
# first. This is sometimes referred to as a "Secondary Sort"
# and can outside streaming mode also be achieved by using a
# combination of a custom Hadoop partitioner with a custom
# equality predicate.
lines = line.split() for line in sys.stdin

for key, values in itertools.groupby(lines, lambda x : x[0]):
  ordered_values = sorted(values, key=lambda x : int(x[1]))

  for value in ordered_values:
    <PIPE value INTO OCTAVE>

    elementid, gaussid = key.split('_')
    nsts, hcf = <OBTAIN RESULT FROM OCTAVE>

    # Emit the result with elementid and gaussid split again
    print "%s\t%s\t%s\t%s" % (elementid, gaussid, nsts, hcf)

<TERMINATE OCTAVE INSTANCE>
```

5.2.4 Export to MySQL

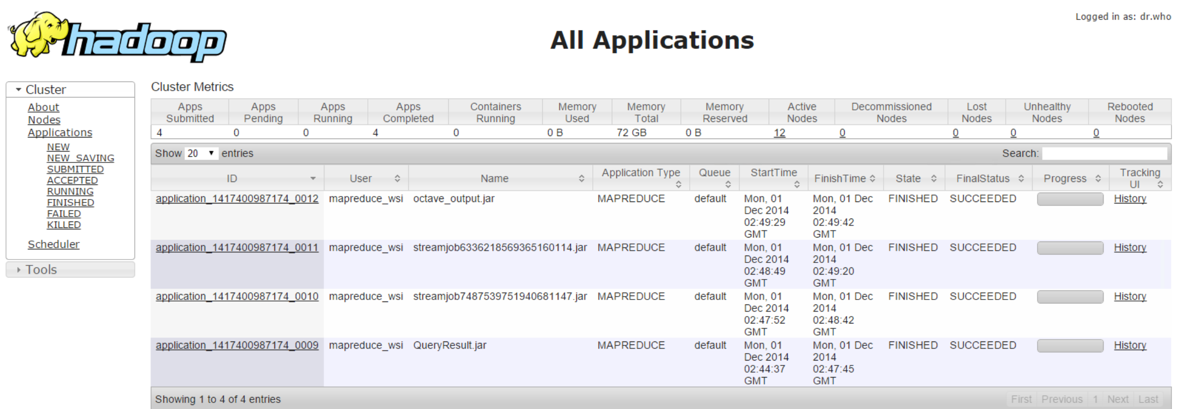
To export the results back to MySQL, the *exportToRDBMS* API provided by MapReduce-WSI is used. No further processing is needed since the resulting tuples already match the format of the destination *octave_output* table (Listing 4.3).

Afterwards, the MapReduce-WSI scope-id is freed and thus all HDFS resources reclaimed.

5.2.5 Logging and Verification

An advantage of Hadoop is that it keeps track of current and previous jobs and exposes web frontends and status pages through which jobs can be examined. Apache Ambari further streamlines the process in that it provides a concurrent view on all the service layers of the Hadoop cluster and their respective status pages.

In our case, running an instance of the MapReduce prototype involves four subsequent MapReduce jobs (two explicitly created, two created by Sqoop). Figure 5.1 shows the Applications Web UI provided by YARN after the prototype completed running. The log files and timings for each stage are still accessible.



Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
4	0	0	4	0	0 B	72 GB	0 B	12	0	0	0	0

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1417400987174_0012	mapreduce_wsi	octave_output.jar	MAPREDUCE	default	Mon, 01 Dec 2014 02:49:29 GMT	Mon, 01 Dec 2014 02:49:42 GMT	FINISHED	SUCCEEDED		History
application_1417400987174_0011	mapreduce_wsi	streamjob6336218569365160114.jar	MAPREDUCE	default	Mon, 01 Dec 2014 02:48:49 GMT	Mon, 01 Dec 2014 02:49:20 GMT	FINISHED	SUCCEEDED		History
application_1417400987174_0010	mapreduce_wsi	streamjob7487539751940681147.jar	MAPREDUCE	default	Mon, 01 Dec 2014 02:47:52 GMT	Mon, 01 Dec 2014 02:48:42 GMT	FINISHED	SUCCEEDED		History
application_1417400987174_0009	mapreduce_wsi	QueryResult.jar	MAPREDUCE	default	Mon, 01 Dec 2014 02:44:37 GMT	Mon, 01 Dec 2014 02:47:45 GMT	FINISHED	SUCCEEDED		History

Figure 5.1: YARN Applications Web UI

Figure 5.2 shows the progress Web UI provided by Hadoop's MapReduce framework during the runtime of the simulation.

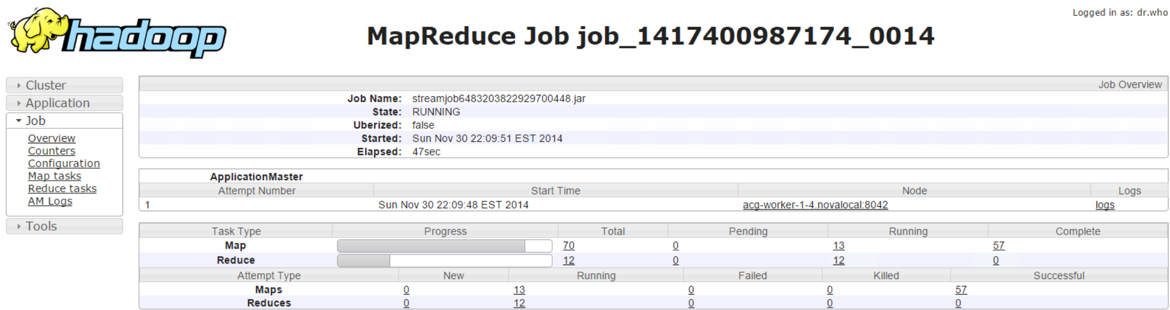


Figure 5.2: MapReduce progress Web UI running the first MR stage

5.3 Baseline Prototype

This section briefly describes the implementation of the baseline prototype against which the MapReduce prototype (Section 5.2) is evaluated. The baseline prototype is a parallel, distributed version of a previous Java prototype written by Reimann et al.[RSM14a]. It takes the same inputs and produces the same outputs as the MapReduce prototype, furthermore it uses the exact same Octave script internally. The difference is that it does not rely on any Hadoop functionality (MapReduce, HDFS) at all. Instead, the prototype manually assigns subsets of work to compute nodes, using each node's local disk to store intermediate results. It also utilizes SQL to filter and aggregate input data directly in the MySQL Cluster database system.

The prototype consists of two different Java programs, referred to as a *SimRunner* and *Worker*. The *Worker* program runs on the nodes in the Hadoop cluster (without using Hadoop functionality or HDFS storage!). Each Worker instance runs the simulation on a subset of the input data respectively. The *SimRunner* executes on the *Admin* node and controls the schedule and allocation of parts of the simulation to the workers.

The algorithm used by the baseline prototype consists of multiple steps, which correspond to the respective steps in the MapReduce prototype.

1. The *SimRunner* determines the size of the input data using a SQL query that retrieves the minimum and maximum FE Element ID and assigns each compute node a subset (specified as a range of FE Element IDs) to process. This is conceptually very similar to the input partitioning performed by Sqoop in the MapReduce prototype.

2. The *SimRunner* uses the `scp` command to copy a JAR archive containing the *Worker* binaries to all compute nodes. It then uses the `ssh` command to run one or more *Worker* instances on each of the compute nodes. In the MapReduce prototype, this corresponds to the MapReduce-WSI functionality to deploy binaries and scripts to the cluster.

3. Each *Worker* uses the SQL query from Listing 5.6 multiple times to fetch input data. This query is a variant of the query shown previously in Listing 4.2. The query is executed once *for each Motion Sequence* (`CURMOTIONSEQ`) and is restricted to the subset of FE Element IDs that is assigned to the worker node (`BEGIN`, `END`). An `ORDER BY` clause is used to sort the result set by (`elementid`, `gaussid`). In the MapReduce prototype, this corresponds to the initial data import using Sqoop (Section 5.2.1) as well as to the first Mapreduce stage, which performs the calculation of the mean variables (Section 5.2.2).

Listing 5.6 Query to extract a worker's subset of the input data

```
SET @CURMOTIONSEQ = /* <motion sequence to be processed> */;

SELECT A.elementid, A.gaussid, NSTS_avg, SIG_V_avg , CNUF_avg
FROM

( SELECT elementid, gaussid, AVG(value) as NSTS_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'NSTS'
  AND elementid > @BEGIN AND elementid <= @END
GROUP BY elementid, gaussid
) AS A,

( SELECT elementid, gaussid, AVG(value) as SIG_V_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'SIG_V'
  AND elementid > @BEGIN AND elementid <= @END
GROUP BY elementid, gaussid
) AS B,

( SELECT elementid, gaussid, AVG(value) as CNUF_avg
FROM pandas_output
WHERE motionid = @CURMOTIONSEQ AND name = 'CNUF'
  AND elementid > @BEGIN AND elementid <= @END
GROUP BY elementid, gaussid
) AS C

WHERE A.elementid = B.elementid AND A.gaussid = B.gaussid
  AND B.elementid = C.elementid AND B.gaussid = C.gaussid
ORDER BY A.elementid, A.gaussid;
```

For each Motion Sequence, a CSV file with the result rows is written. Due to the ordering by `(elementid, gaussid)`, the rows in each of the individual CSV files correspond to each other (recall that each Motion Sequence uses the same FE Elements and Gaussian

Points!). The Worker then does a sequential scan through all CSV files in parallel and joins the lines. This split-sort-and-merge approach is required since the CSV files do not necessarily all fit into main memory. The Worker then uses the mapping from hours-of-day to Motion Sequence IDs to generate an ordered list of values over the course of a Daily Activity of 24 hours. This list is then forwarded to an instance of Octave that runs in parallel and computes the final results. This stage corresponds closely to the second Mapreduce stage of the MapReduce prototype (Section 5.2.3). CSV files are afterwards deleted.

4. Each *Worker* emits a SQL update to write their results back into the `pandas_output` table. This corresponds to the final data export from HDFS to MySQL in the MapReduce prototype (Section 5.2.4).

5. The *Runner* waits until all of the Workers have completed their work.

6 Evaluation

This chapter evaluates the performance of the MapReduce prototype against the baseline prototype presented earlier. As a preliminary, challenges for evaluating a system as complex as the bone simulation in the context of a Hadoop cluster are presented and discussed. Furthermore, the cluster topology outlined in Chapter 3 is augmented with the exact physical resources available to the simulation.

6.1 Generating Test Data

The input to the simulation is given as a database table with a scheme as presented in Listing 4.1 (*pandas_output* table). The contents of this table can either be generated by running the PANDAS simulation on real-world input data, or by using artificially generated test data. For this thesis, the latter approach was chosen.

To be able to measure the performance of the simulation, artificial test data needs to be generated such that the runtime of the simulation with artificial inputs accurately reflects the time it would take to run on actual data. Luckily, the part of the simulation that is to be brought onto MapReduce only involves arithmetic operations. Its runtime does not depend on any characteristics of the input values, nor are there any dependencies or requirements for these values. Thus, the contents of the *pandas_output* table (i. e., the *value*) column) can be drawn from a random number generator.

The full Java code to generate test data is found in Appendix C. Since the size of the input is a variable during evaluation, it is left configurable. The corresponding MySQL table uses the InnoDB storage format with the server defaults.

6.1.1 Defining standard test data sets

For the evaluation, three different input test sets are defined and generated once:

1. *Small*: For this data set, the input table is populated with the following cardinalities (see Section 4.2)

`motionid`	1
`elementid`	1000
`gaussid`	8
`timestep`	100
`name`	22

In total, these are 17.6 million rows. The size of the database as reported by MySQL (counting both data and index/key size) is 1141 MiB.

2. *Medium*: For this data set, the input table is populated with the following cardinalities (see Section 4.2)

`motionid`	2
`elementid`	2000
`gaussid`	8
`timestep`	200
`name`	22

In total, these are 140.8 million rows. The size of the database as reported by MySQL (counting both data and index/key size) is 9138 MiB.

3. *Large*: For this data set, the input table is populated with the following cardinalities (see Section 4.2)

`motionid`	3
`elementid`	3000
`gaussid`	8
`timestep`	300
`name`	22

In total, these area 475.2 million rows. The size of the database as reported by MySQL (counting both data and index/key size) is 30806 MiB.

The tables use no compression, but an optimized storage format using smallest-possible integer types (see Section 4.2).

6.2 Challenges with Hadoop

The fault-resistant, self-optimizing architecture of the Hadoop ecosystem makes it easy to accidentally produce skewed measurements. The following scenarios would all cause performance degration or incorrect measurements:

- HDFS operation close to total storage capacity: This affects the number of times new files are replicated across HDFS `DataNodes`. If the same data is accessed from multiple nodes, read/write performance is sensitive to the number of times each block is replicated across the filesystem. If single nodes becomes overloaded, unexpected side-effects may occur, in particular if HDFS and YARN run on the same nodes (which is the case in our setup).
- Worker failure: Due to the fault-tolerant architecture of Hadoop, worker failure easily goes unnoticed since the part of the task that is allocated to the node which failed gets transparently re-allocated to the remaining nodes. This re-execution incurs significantly increased computation cost (unless the effect is negligible due to sheer cluster size), but does not affect the correctness of the result.
- Insufficient scale: Generally, YARN and MapReduce do not distribute workloads unless the size of the task exceeds a certain threshold. This makes sense: the constant cost of setting of slots across the cluster is very high and involves coordination, copying mapper and reducer binaries and other working data associated with the MapReduce task. Furthermore, Shuffling and merging of reduced results incurs even more network overhead. Unfortunately, the default heuristic to decide is the size of the input data, which is not necessarily a measure of the size of the task.

6.3 Challenges due to Virtualization

Virtualization of computation resources makes it harder to accurately measure performance metrics since an inconvenient mapping from virtual to physical resources can become a bottleneck. Possible problems include:

- Over-allocation of underlying resources. It is economical in virtualized environments, in particular in commercial clouds which charge their users money for their services, to over-allocate resources based on the observation that most usage patterns include a relatively low average usage with occasional spikes (i .e, Web Services). This can become a problem if the computation intends to fully utilize the resources it is is allocated to.

  In particular, a single physical host can be over-allocated in several ways, including but not limited to:

  1. CPU: If fully utilized virtual CPU cores (“vCore” in OpenStack terminology) across multiple VMs map to a smaller number of physical CPU cores, the hypervisor’s process scheduler will time-slice the physical CPU cores between the virtual CPU cores, effectively throttling each VM. Additionally, CPUs that support Hyper-Threading

or related technologies[1] do “virtualize” their cores on a lower level in that they advertise two full cores for each physical CPU core to the hypervisor operating system, switching back and forth between two execution contexts when one of the two stalls on a memory access. While this increases performance, it is not as fast as two real CPU cores, thus making CPU cores and even more virtualized CPU cores a generally unreliable metric for computation power. However, many real-world workloads are not CPU bound. MapReduce in particular trades increased IO (since intermediate output between map and reduce stage is written to disk) for the ability to parallelize workloads easily.

2. Network: Commodity network interfaces provide 1 Gigabit/s (full-duplex) bandwith, which amounts to an effective bandwith of 100 MB/s after TCP/IP packaging overhead is subtracted. Since many Network Storage solutions are able to serve a single file with such rates (in particular the threefold replication that HDFS defaults to is easily able to achieve a 100 MB/s reading rate, even if the data comes from 5400 RPM disks), it follows that multiple VMs on the same physical host network interface can exceed the available incoming or outgoing bandwith. The degree to which this is a concern depends on the IO pattern observed by the computation, as well as on the performance of the network storage solution that is deployed in the cluster.

- Non-realistic network latency if the majority of virtualized nodes reside on the same physical server. In this case, nodes communicate via regular TCP/IP through the virtual Ethernet network that connects them, but the VM hypervisor recognizes the traffic as local and passes it through the local loopback interface, or an even more efficient IPC mechanism. This makes network communication (i. e., RPCs) seem more efficient that it would be in a truly physically distributed environment. While this is good for performance, it affects how well results generalize to (larger) clusters with a different topology.

6.4 Virtual Cluster Configuration

This section describes the configuration of the virtual cluster in which the evaluation thas taken place. There were two major considerations that drove the decisioning on cluster configuration:

1. We want to scale the physical resources that are available for this thesis close to their maximum extent and perform measurements at peak capacity.

[1] http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

2. We want to avoid the challenges and problems outlined in previous sections 6.3 and 6.2. In particular, we need to achieve a near one-to-one mapping from virtualized to physical resources with no or very little over-allocation, using multiple physically separated hosts while ensuring that sufficient spare capacity is available both for monitoring and for file system operation.

The virtual cluster contains 8 Hadoop nodes.

6.4.1 Physical Hardware

The entire virtual cluster runs on a computing cluster with 8 interconnected hosts, each of which provides the following hardware:

- 2 x Intel Xeon E5-2630 2,3GHz (12 Cores, 24 with HyperThreading)
- 16 x 16 GiB RAM (256 GiB)
- 10 Gbs Ethernet Controller

Two such hosts were exclusively available for the work on this thesis. Three other hosts were available for use, but shared with other users of the cluster (but not under heavy load or close to capacity). The three remaining hosts were unavailable. When creating a VM in the OpenStack UI, one can specify an availability group for it. Since availability groups can be restricted to specific physical hosts, this allows "locking" VMs to specific hosts. In this work, three availability groups are used for this purpose: *Host7* and *Host8*, which refer to the hosts which were used exclusively and *Host3_4_5*, which spans the three shared hosts.

Reliable storage is accessible via a StorWize V7000 *Storage Attached Network* (SAN), a separated network providing access to disk arrays which are mapped to appear as if they were locally attached devices. The total SAN storage capacity is about 9 TiB of redundant storage. For the work on this thesis, about 5 TiB were available. The SAN is connected to the computing cluster through a 8 Gb/s Fiber Channel connection, however the total bandwith to the SAN may be lower due to various implementation details.

6.4.2 Mapping the Virtual Cluster to Physical Resources

This section describes the mapping from virtual cluster components to physical cluster resources. For comparison, refer to the (logical) topology of the virtual cluster back in Figure 3.3.

Each of the two physical hosts that we use exclusively (i .e, *Host7* and *Host8*) host 4 identical Hadoop nodes for a total of 8 Hadoop nodes. Furthermore, our MySQL cluster instance utilizes 4 VMs as data nodes, 2 placed on *Host7* and 2 placed on *Host8*. Each of the dedicated physical hosts also runs a MySQL cluster SQL node. The Apache Ambari management node *Ambari* and

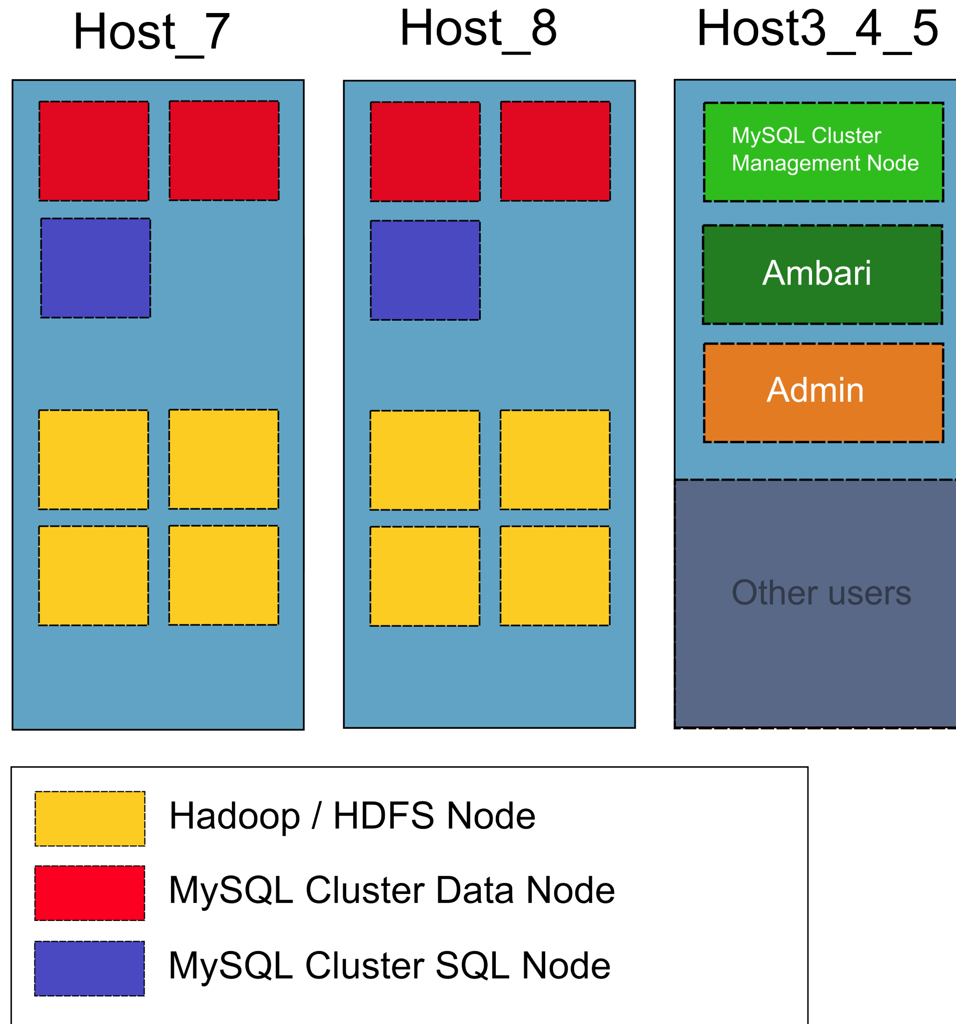


Figure 6.1: Physical Topology of the Virtual Cluster

the Windows Server 2012 node *Admin* run in the (shared) *Host3_4_5* availability zone. This zone also hosts the MySQL cluster management node. All of the services in this zone are not under heavy load during the simulation, so having them in a shared environment should not affect results.

The entire setup is visualized in Figure 6.1. Note that each colored box refers to a full VM and not only a service instance. Each VM offers all the services required for their role in the cluster, for example a Hadoop node hosts services specific to YARN, MapReduce and HDFS as well as instrumentation and measurement services that are used by Ambari to collect cluster health data (see Section 3.4.4).

6.4.3 Hadoop Cluster Nodes

The virtual hardware allocated to each of the 8 nodes in the Hadoop cluster is shown in the following table:

Operating System	CentOS 6.5
VCores	4
RAM	8 GiB
Local Disk	20 GiB
Network Disk (SAN)	120 GiB

Because *Host7* and *Host8* each hold 4 of those nodes, the total resource allocation for each of these physical hosts is:

VCores	16
RAM	32 GiB
Local Disk	80 GiB

This achieves a one-to-one mapping of virtual CPU cores to (advertised) physical CPU cores. The amount of RAM allocated is less than a fourth of the available 256 GiB, however this was deliberately chosen to ensure that the working set of the simulation does not fit into main memory, which could skew results.

HDFS Sizing

Each Hadoop worker node hosts a HDFS *DataNode* which is allowed to use up to 100 GiB of the attached storage device (reserving about 20 GiB for other purposes, e.g. local spill space, file system overhead). With 8 nodes, this amounts to 800 GiB of raw HDFS space across the Hadoop cluster. Since HDFS is configured to use three-way replication for storing files, only a third of the 800 GiB are effectively available.

6.4.4 MySQL Cluster Data Nodes

The virtual hardware allocated to each of the 4 data nodes in the MySQL cluster is shown in the following table:

Operating System	CentOS 6.5
VCores	8
RAM	64 GiB
Local Disk	20 GiB
Network Disk (SAN)	0 GiB

A network disk storage quota was not required since the cluster operates on *Diskless* mode, i. e., it holds the database tables and indexes in main memory only. The local disk storage is required to run the operating system.

The MySQL Cluster data nodes and the Hadoop nodes do share CPU resources, however the main purpose of the MySQL cluster data nodes is to hold table rows in memory and to retrieve them upon request of a SQL node. This means the CPU requirements are relatively low and the effect of "stealing" CPU cores from the Hadoop nodes should be negligible. During heavy query load, we never saw the CPU load of any of the data nodes exceed 5%.

Configuration

Our MySQL Cluster's performance was found to be very sensitive to configuration. To make our results reproducible, the following is a commented version of the `ndbmtd` configuration we ended up using:

Listing 6.1 MySQL Cluster configuration (`ndbmtd`)

```
# Each data node uses 30 GiB for data storage.
DataMemory=30G

# Each data node uses 8 GiB to store indexes.
IndexMemory=8G

# 2 replicas, this means 2 of the 4 data nodes
# hold copies of the other pair.
#
# Thus the effective storage is 60 GiB of data
# and 16 GiB of indexes.
NoOfReplicas=2

# Number of concurrent operations (i.e., row modifications)
# Our cluster is able to handle high numbers since each
# data node has 8 vCPUs. However, each operation record
# consumes ~100 KiB main memory.
MaxNoOfConcurrentOperations=500000
MaxNoOfLocalOperations=550000

# Use at maximum 8 threads to serve data requests. This
# fits the 8 vCPUs allocated to the VM.
MaxNoOfExecutionThreads=8

# Run from main memory only.
Diskless=1

# Locking pages in memory (using the POSIX mlock() API)
# prevents them from being accidentally swapped out
# to disk, which could be another source of unexpected
# slowdown or side effects.
LockPagesInMainMemory=1
```

All database tables were further created with the `ndb_table_no_logging` property turned on because MySQL Cluster with the `Diskless=1` flag still writes operation logs to disk unless the aforementioned property is set. To test the setup, we measured global disk IO across the cluster (using the SAN's storage console) and verified that SQL updates and queries do not hit the disks.

6.4.5 Testing the Hadoop Cluster

To verify that the functionality of the Hadoop cluster is all in place, we ran the *terasort* benchmark[2] on a data set which was randomly generated using the *teragen* tool. Results were then verified using the *teravalidate* program.

We found the benchmark working properly. However we measured that the combined write throughput of the SAN while running the benchmark is only in the range of 50 MiB/s while CPU utilization in the cluster is consistently below 100%. This likely indicates that disk input/output will be the bottleneck for many real-world scenarios.

6.5 Optimizing MySQL Performance

Both prototypes rely on executing SQL statements. The baseline prototype performs the entire data aggregation step using SQL and the MapReduce prototype uses Sqoop to run parallel SQL queries from MapReduce mappers. It is not the purpose of this thesis to squeeze the maximum possible performance out of these queries, however insufficiently optimized SQL queries can easily run a magnitude slower than optimized queries. A difference in magnitude is enough to skew results for the entire simulation. Thus to achieve realistic measurements, all queries were briefly investigated and optimized. It should be noted that we only exploited straightforward and widely applicable optimization strategies requiring no expert knowledge.

All relevant queries in the simulation operate on the *pandas_output* table. This table uses a composite primary key, which means the natural storage order of the rows provides an index which spans all columns of the composite primary key. This index is used for queries involving a predicate (`WHERE`-clause) on all the columns in the composite primary key (with some restrictions on ordering relations such as less-than or greater-than). The index also matches any predicate on *a left prefix of the columns*, for example a query that only restricts the `motionid` and the `timestep` column.

6.5.1 Optimizing Baseline Query

The baseline prototype as described in Section 5.3 relies on a variant of the SQL query in Listing 5.6. This query does not utilize this default index because the `timestep` column is not part of the predicate (`elementid` and `gaussid` are however implicitly part of the predicate since they are referenced from the `GROUP BY` clause[3]). This is confirmed by using the `EXPLAIN` SQL statement with the query.

[2]https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html
[3]http://dev.mysql.com/doc/refman/5.0/en/group-by-optimization.html

Our hypothesis was that changing the order of the columns such that `timestep` comes last increases performance of the query. This is because the predicate then becomes a left prefix of the composite key and the default index can be used. The same effect could be achieved by creating an additional index on the table that omits the `timestep` column, however this would significantly increase storage requirements (table size is already dominated by the size of the keys, so indexes become large as well).

To verify the hypothesis, we used the *Small* test data set as described in Section 6.1.1. We then ran the query from Listing 5.6 with a `LIMIT` of 20 against a local instance of MySQL and measured the execution time of the query as reported by MySQL itself. We averaged the results of three runs. The results of the experiment are shown in the following table:

Column Order	Query time (s)
`motionid, timestep, elementid, gaussid, name`	173.29
`motionid, elementid, gaussid, name, timestep`	123.5

Confirming the hypothesis, a measurable speedup is achieved by putting the `timestep` column last. Furthermore, using `EXPLAIN` on the queries verifies that the faster query indeed uses the index. All further evaluations were carried out using this modified column order.

Column vs. Row Store

This dependence on column order to match indexes such that columns can be efficiently selected and then aggregated raises the question whether a column store could be the preferred storage format for the data. While open for future evaluation, several arguments weight in favour of row-based storage systems:

- Of the 6 columns in the database, 3 (`elementid`, `gaussid`, `value`) are selected. Using a column store thus reduces the amount of data that must be fetched by 50% at maximum.
- While not part of this thesis, the `pandas_output` table is populated by the PANDAS Web Service, which itself is a complex system that is not easy to migrate to a different data storage system (by comparison, creating additional indexes or changing column order is a modification that is easy to incorporate). Further PANDAS naturally consumes and produces data in rows, which rather argues for a row-based storage than for a column-based one.

6.5.2 Optimizing Sqoop Import Query

The MapReduce prototype uses the Sqoop query from Listing 5.1 to import data into HDFS, specifying the `elementid` column as split column along which data is partitioned. The query itself only places a predicate on the `name` column. However, Sqoop transparently inserts a second predicate on the `elementid` column to select the subset of the input data assigned to each worker. This means, the query does not utilize the default index either since the `motionid` and `gaussid` columns are not part of the predicate. Reordering columns again would certainy help, however putting `motionid` first is partly motivated by the way how the previous PANDAS simulation is parallelized and putting `gaussid` after `elementid` is required to efficiently evaluate the `GROUP BY` clause in the query from Listing 5.6. Thus, we decided to add an index on the `elementid` and `name` columns only. Since this index only contains two columns, its storage requirements are less severe than for the default index. Our hypothesis was that this increases Sqoop import performance.

To verify the hypothesis we used the *Medium* test data set as described in Section 6.1.1. We then ran the MapReduce prototype and measured Sqoop import time only using 8 Sqoop mappers. We averaged the results of three runs. The results of the experiment are shown in the following table:

Schema	Query time (s)
Default index only	303.6
Index on (`elementid`, `name`)	99.4

Further, we measured a peak network throughput of 344 Mb/s on the SQL node of the cluster with the additional index as compared to 72 Mb/s without the index (measured using the `iftop` program). Confirming the hypothesis, a 3x speedup is achieved by adding the index on the `elementid` and `name` columns. All further evaluations were thus carried out using this index.

6.5.3 Optimizing Sqoop Boundary Query

As described in Section 2.5.2, Sqoop initially queries the MySQL database to determine the minimum and maximum values along the split column in order to partition the input data among worker slots. Since this query is auto-generated by default (and tweaked by MapReduce-WSI as needed), it is not necessarily optimal. In the case of the import query from Listing 5.1, it requires a full table scan since the MySQL query optimizer does not know that the range of FE Element IDs does not depend on the `name` column.

However, the simplest possible query to get the intended result is

```
SELECT MIN(elementid), MAX(elementid) FROM pandas_output
```

Note that this query is able to utilize the index on the `(elementid, name)` columns that was added as a result of the measurements in the previous Section 6.5.2. Without this extra index, the same theoretical effect on the performance of the query can be achieved by fixing the first column (e.g. `motionid=0`). This would not affect the result because the range of FE Element IDs is the same for each Motion Sequence. Adding this predicate, the default index would match.

Our hypothesis was that using this custom boundary query increases Sqoop import performance drastically since the result is directly available from the index and no full table scan is required. To verify the hypothesis we used the *Medium* test data set as described in Section 6.1.1, having added the index on `(elementid, name)`. We then ran the MapReduce prototype and measured Sqoop import time only using 8 Sqoop mappers. We averaged the results of three runs. The results of the experiment are shown in the following table:

Boundary Query	Query time (s)
Default	99.4
Improved query	61.0

Confirming the hypothesis, a 30% speedup to the entire Sqoop import is achieved by using the optimized boundary query. Note that the boundary query itself is now in the magnitude of a second or less since it is directly answered from the index. All further evaluations were thus carried out using the optimized boundary query.

6.6 Measuring MapReduce vs. Baseline

In this experiment, we measured the performance of the MapReduce implementation against the baseline implementation. Input and output data are stored in the MySQL Cluster instance.

Independent variables (variables varied) for this experiment were:

- Size of the input data. We tested with three different artificial data sets generated as described in Section 6.1.1.
- Number of nodes and jobs per node participating in distributed Octave computations. The number of nodes and jobs per node multiplied together gives the total level of parallelism for running the simulation. We tested with three different levels of parallelism. For the baseline implementation this we directly controlled the number of worker nodes being used and also for each node how many instances of the *Worker* program are run on it in parallel. For the MapReduce implementation we configured the number of map and

reduce jobs that are launched and how many resources (CPU, RAM) each job consumes. Putting restraints on the resources is a simple (yet indirect) way of limiting the number of concurrent mapper or reducer jobs running on the same node. It is important to note that this is only an upper bound though. The exact number of jobs that are launched can be lower and is at YARN's discretion. Also, there is no control about how many nodes are effectively used in parallel since YARN factors the size of the input data into account. In our setup we empirically verified that our Hadoop settings resulted in the correct number of concurrent jobs (+-1) being launched. The follow shows for each of our configurations also the configuration parameters we passed to Hadoop.

1. *Parallelism #8 (4 Nodes, 2 job each)*

```
mapreduce.job.maps=8
mapreduce.job.reduces=8
mapreduce.map.memory.mb=4000
mapreduce.reduce.memory.mb=4000
```

2. *Parallelism #16 (8 Nodes, 2 jobs each)*

```
mapreduce.job.maps=16
mapreduce.job.reduces=16
mapreduce.map.memory.mb=2200
mapreduce.reduce.memory.mb=2200
```

3. *Parallelism #32 (8 Nodes, 4 jobs each)* (Assigns one job to each CPU in the cluster)

```
mapreduce.job.maps=32
mapreduce.job.reduces=32
mapreduce.map.memory.mb=1100
mapreduce.reduce.memory.mb=1100
```

Dependent variables (variables measured) were:

- *aggregation_time*: Time taken for aggregating (averaging) inputs. This includes importing inputs from the `pandas_output` table in the MySQL Cluster. It also includes any setup costs (i. e., copying binaries or mapper/reducer scripts).
- *octave_time*: Time taken for performing the Octave simulation. This includes exporting results into the `octave_output` table in the MySQL Cluster. It also includes any setup costs (i. e., copying binaries or mapper/reducer scripts).
- *total_time*: Total simulation runtime, which is simply the sum of the two previous variables.

Time measurements are taken by the *Admin* node. For the baseline prototype, the ratio between the *aggregation_time* and *octave_time* variables is self-reported by each Worker node, applied to the *total_time* measured by the *Admin* node and then the maximum reported time of all the Worker instances is taken.

6.6.1 Hypotheses

Our hypotheses for this experiment were:

1. The MapReduce prototype has constant additional initial cost for setting up and scheduling MapReduce jobs on the cluster. The baseline implementation has very little setup cost by comparison, since it only copies a (very small) binary to all the distributed workers and launches it there.
2. Both implementations scale linearly with respect to the level of parallelism. This means we expect that jobs runs twice as fast if the level of parallelism is doubled, all other settings fixed.
3. Both implementations scale at least linear (i.e, linear or better) with respect to input data size given a fixed level of parallelism. Linear scaling means that we expect that processing a data set of twice the size of the previous data set takes no more than twice the original time to run. In terms of raw row count, the *Medium* data set is 8x the size of the *Small* data set and the *Large* data set is 27x the size of the *Small* data set, so we would expect 8x respectively 27x the runtime of the *Small* data set. However, this is only a upper bound because each of the stages of the simulation reduces one dimension of the input, thus significantly reducing the size of the working set as the workflow progresses. After the data aggregation stage, the `timestep` column is reduced and after the first part of the Octave calculation the `motionid` column is reduced as well. The runtime of the numeric Octave simulation itself and the size of the final `octave_output` table then only depend on the cardinality of the `elementid` column!

   Thus, due to the reducing nature of the simulation we expect the effective scalability in the *total_time* variable with respect to input data size and a fixed level of parallelism to be significantly better than the upper bound we gave.

   We expect the *octave_time* variable to be dominated by the numeric Octave simulation and thus to scale linearly with the `elementid` column. We expect the *aggregation_time* variable to be dominated by the cost of fetching input rows and thus to scale linearly with the size of the data set.

6.6.2 Results

The following tables show the results for each of the dependent variables. The first value in each cell refers to the MapReduce prototype and the second value refers to the baseline prototype respectively. All measurements were taken using the average of three iterations.

*	#8	#16	#32
Small	64.14 / 29.07	62.33 / 21.12	66.62 / 16.48
Medium	158.21 / 216.39	117.71 / 141.38	110.24 / 117.62
Large	472.96 / 733.19	295.51 / 491.99	243.86 / 400.20

Table 6.1: Measurements: *aggregation_time* (seconds)

*	#8	#16	#32
Small	169.49 / 27.73	77.18 / 16.09	72.77 / 12.51
Medium	156.67 / 56.87	102.20 / 30.46	86.35 / 26.63
Large	209.62 / 83.80	124.59 / 45.19	102.64 / 35.94

Table 6.2: Measurements: *octave_time* (seconds)

*	#8	#16	#32
Small	207.86 / 56.81	139.51 / 37.21	139.40 / 29.00
Medium	314.89 / 273.26	219.91 / 171.84	196.60 / 144.25
Large	682.58 / 816.99	420.10 / 537.18	346.50 / 436.15

Table 6.3: Measurements: *total_time* (seconds)

The following diagram shows the *total_time* variable with respect to different levels of parallelism (left-to-right) and different input data sizes (front-to-back).

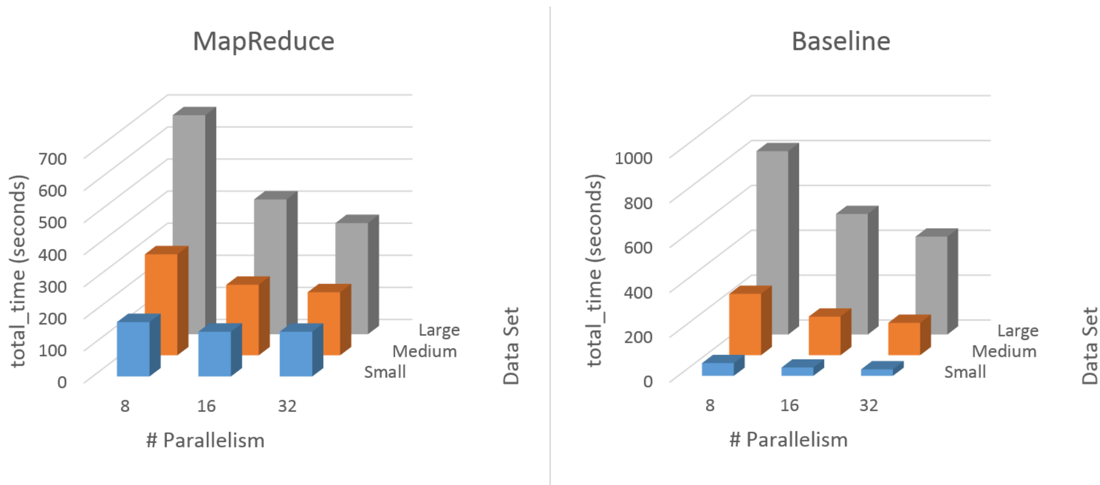


Figure 6.2: MapReduce vs. Baseline Prototype Evaluation Results

The following diagram shows the *aggregation_time* and *octave_time* variables with respect to input size, both prototypes run with a fixed level of parallelism of 32.

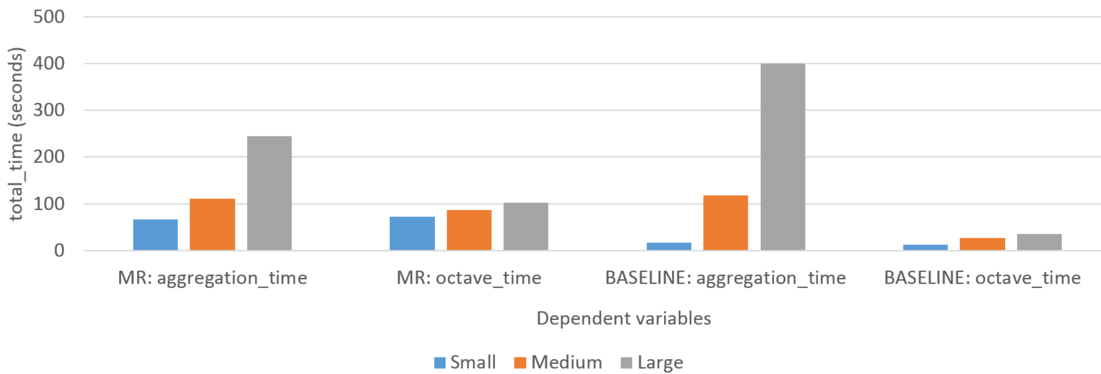


Figure 6.3: Results with Respect to Input Data Size, Parallelism #32

The following diagram compares the two prototypes directly.

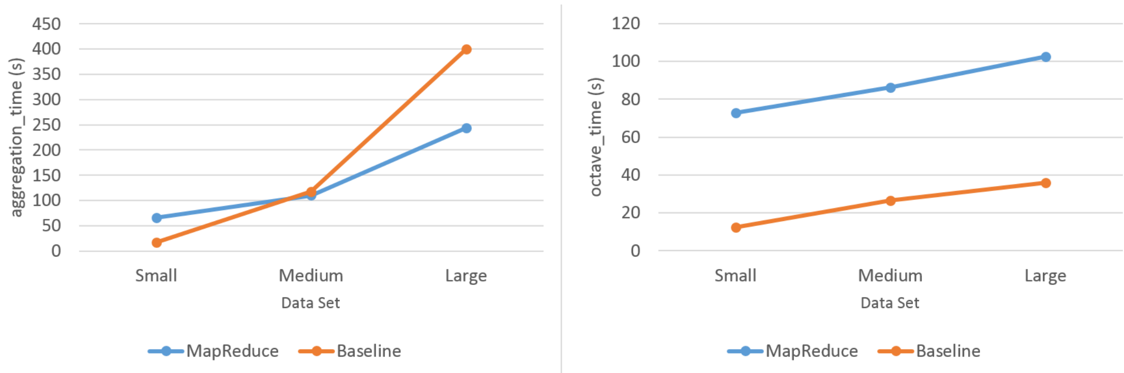


Figure 6.4: Comparison of the Prototypes, Parallelism #32

The following diagram shows the same variables, but both are *normalized by the actual input size of each stage*. Normalization is relative to input size with the *Small* data set. This makes it easier to reason about scalability: if a prototype has negligible constant cost and scales linearly with respect to the metric by which the diagram is normalized, all bars should be of approximately the same height.

In this case, the *aggregation_time* variable for the *Medium* data set is divided by 8 since it input size is 8x as large as the input for the same stage in the *Small* data set (2x the number of FE Elements, 2x the number of Motion Sequences, 2x the number of timesteps). The *octave_time* variable for the *Medium* data is however only divided by 4 since the timesteps have already been reduced at this stage.

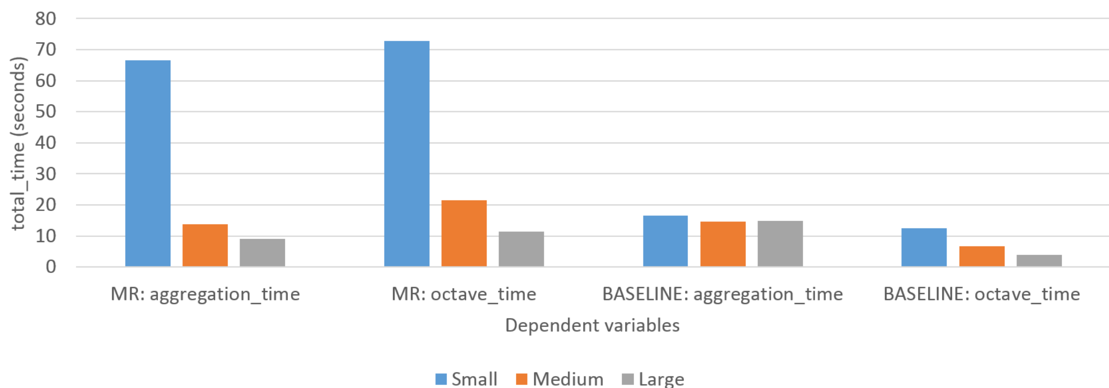


Figure 6.5: Results with Respect to Input Data Size (Normalized), Parallelism #32

The following diagram is similar but *normalizes by the actual output size of each stage*. As an example, the *aggregation_time* variable for the *Medium* data set is divided by 4 since its output

size is 4x as large as the output for the same stage in the *Small* data set (2x the number of FE Elements, 2x the number of Motion Sequences). The *octave_time* variable for the *Medium* data is however only divided by 2 since the Motion Sequences are reduced during this stage.

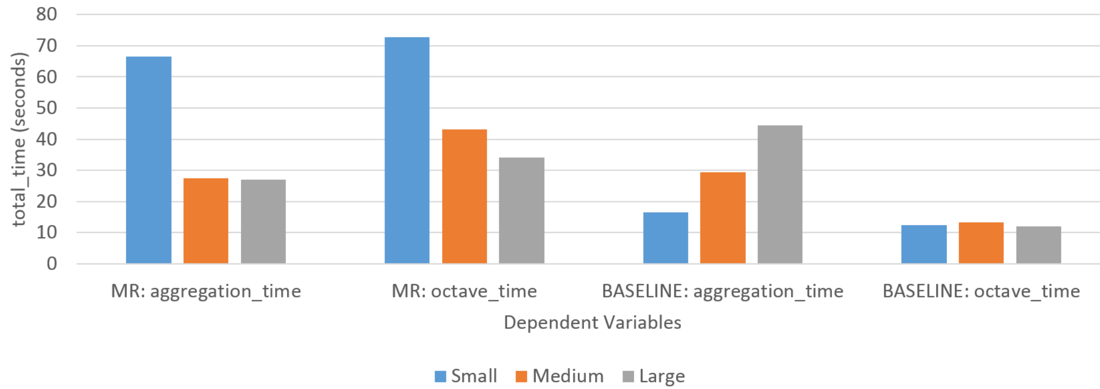


Figure 6.6: Results with Respect to Output Data Size (Normalized), Parallelism #32

6.6.3 Discussion of Results

With respect to the original hypothesis, our findings were:

1. Figure 6.2 supports the argument since all towers in the left diagram (MapReduce prototype) have a constant socket as compared to the right diagram (baseline prototype). Figure 6.5 and Figure 6.6 also suggest that this hypothesis is correct: The fact that the MapReduce prototype is inefficient running the *Small* data set in comparison to both the larger data sets and the results obtained by the baseline prototype can be explained by a high setup cost for MapReduce jobs, which amortizes as the size of the data increases. It would be important to measure with larger data sets to further solidify the hypothesis.

2. The prototypes do scale with increasing level of parallelism in the cluster, but the measured scalability is in both cases worse than linear. This can be seen by calculating the difference between the *total_time* for the different levels of parallelism (data size: *Large*). This has the advantage that constant cost is factored out.

*	#8 - #16	#16 - #32
MapReduce	262.48	73.6
Baseline	279.81	101.03

With linear scalability, the increase in the second column should be half the value of the first column. As the table shows, this is not the case, thus scalability of the prototypes is worse than linear.

3. We find our upper bound hypothesis confirmed in that both implementations show better-than-linear scalability in the *total_time* with respect to total input data size. This can be seen in Figure 6.3 (the increase in runtime is less than 8- respectively 27-fold).

   Figure 6.6 shows that the *octave_time* of the baseline prototype is indeed linear with respect to the cardinality of the `elementid` columns, hinting that the runtime of the Octave simulation dominates the runtime of the split-sort-and-merge algorithm where the `motionid` column is reduced. The same can be observed for the MapReduce prototype if the setup cost is subtracted.

   Figure 6.5 shows that the *aggregation_time* of the baseline prototype is indeed linear with respect to input size. This makes sense since data aggregation is naturally dominated by the cost of fetching input data. The MapReduce prototype shows the same behaviour. This can be seen in the left half of Figure 6.4, where both graphs triple their slope at the mid-point.

Independent of our original hypothesis, several other key observations can reasonably be drawn from the measurements:

1. The left side of Figure 6.4 shows that data aggregation in the baseline prototype has an overhead of about 3x compared to the MapReduce prototype. A possible explanation for this deficiency of the baseline prototype is found in the natural limitations of aggregating data in MySQL Cluster (see Section 2.5.2). While the data nodes are able to achieve high parallel throughput, the calculation of the mean effectively happens on a single SQL node which can easily become overloaded and suffer from congestion effects. The MapReduce implementation of the data aggregation by comparison has no dependence between the nodes after the initial data import.

2. There is a difference in `octave_time` per FE Element of the MapReduce prototype and the baseline prototype with otherwise identical settings. This can be seen as a slightly different slope of the graphs in the right half of Figure 6.4. However, both prototypes execute the exact same Octave calculation. There are three possible explanations as to what the cause of this is:

   - The overhead of using HDFS as compared to the local file system in the baseline split-sort-and-merge approach. While the former necessarily includes disk writes and network traffic, it is likely that the latter never hits any disk since the size of the input data at this stage of the simulation is small enough to fit the respective VM's page cache. If true, the effect should be mitigated with a larger data size.

   - The cost of exchanging lines of data between the prototype and the Octave instance. Both prototypes implement this coupling through the same mechanism (a bi-directional pipe). However, the baseline prototype accesses the pipe from Java while the MapReduce prototype uses Python. Preliminary measurements running

only the coupled Octave simulation on 1000 elements showed that Python's IO handling is indeed slower than Java's so this likely contributes to the effect we saw.

- YARN's scheduling decisions and the fact that we were not able to configure the mapping from jobs to nodes as well as we were able to do for the baseline prototype. Looking at the MapReduce logs we found that in some cases the reduce time increased because the *ApplicationMaster* MapReduce master job takes up enough resources in the cluster such that YARN does not schedule all the reducers at the same time. The remaining reducers (usually one or two) then run sequentially after all reducers in the first batch have finished. It is possible that this affected some of the measurements taken for the MapReduce prototype. Figure 6.7 shows the CPU usage across the entire cluster when running the MapReduce prototype on the *Medium* data set using a level of parallelism of 32. Cluster CPU usage is measured and visualized by Ambari. During most of the time, the cluster is almost fully utilized. However, at about two thirds, CPU usage drops. This is when only one or two reducers continue to run while the rest of the work is already done. It should be noted that this problem can be coped with by tweaking the configuration of the MapReduce job, most easily by using more and smaller mapper and reducer jobs.

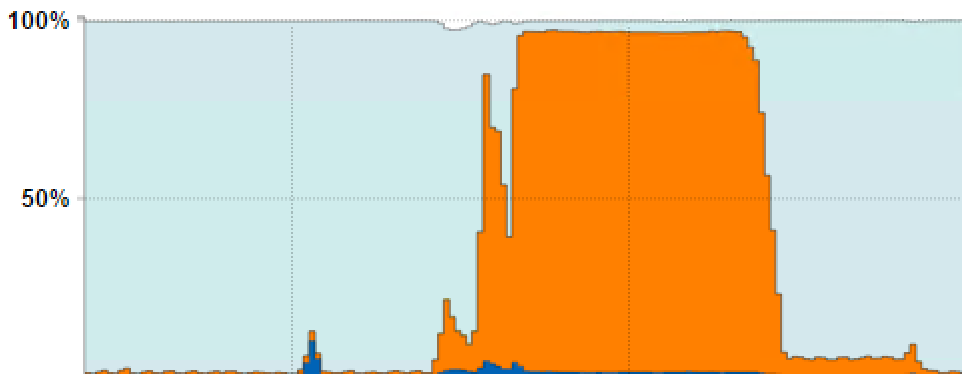


Figure 6.7: Cluster CPU Usage with Sub-optimal reducer Scheduling

To conclude, both prototypes scale to our expectations with respect to input size. However, both prototypes show worse-than-linear scalability with respect to the level of parallelism in the cluster. The MapReduce prototype has large constant cost but finally outperforms the baseline prototype for the *Large* data size (346.50s vs. 436s with a level of parallelism of 32). As our result shows, this is because the data aggregation step performs better with the MapReduce prototype than with the baseline prototype.

7 Workflow Integration

While the MapReduce and the baseline prototype have been developed in Java, both are only surrogates for an implementation in a workflow language such as WS-BPEL. In this Chapter a concept for implementing the MapReduce prototype in WS-BPEL using the utilities provided by the SIMPL framework (see Section 2.2.2) is outlined.

7.1 Formulating the MapReduce prototype using WS-BPEL

For reference, the source code of the Java-based MapReduce prototype can be found in Appendix B. If all measurement-specific logic is ignored, the prototype executes the following steps in fullfilment of Section 4.4:

- (Re-)create the *octave_output* table to hold results.
- Load mapper and reducer Python scripts from disk.
- Allocate a MapReduce-WSI scope-id.
- Submit 4 jobs to the cluster: Sqoop import, first Mapreduce stage, second Mapreduce stage and eventually data export.
- Free the MapReduce-WSI scope-id.

This sequence of steps directly maps to WS-BPEL activities (*Assign* activities to set arguments omitted):

- The SIMPL *IssueCommand* data management activity provisions the *octave_output* table.
- mapper and reducer scripts are either embedded into the workflow or fetched from disk using SIMPL's *RetrieveData* data management activity.
- An *Invoke* activity is used to call the MapReduce-WSI *createScope* API.
- For each of the jobs another *Invoke* activity is used to call a MapReduce-WSI API.
- An *Invoke* activity is used to call the MapReduce-WSI *deleteScope* API.

8 Conclusion

In the early chapters of this thesis, key topics were presented. This included a brief overview of Web Service and workflow technology in general. Furthermore, the multi-scalar bone simulation as the main subject of this thesis has been introduced. Important Big Data concepts, in particular the MapReduce paradigm, its implementation as part of Apache Hadoop and NoSQL systems in comparison with classic database systems were explained. We also briefly described current advances in computing cluster design, in particular the role which virtualization plays.

In Section 4 we derived the exact control and data flow of the coupling workflow from previously published work by Reimann et al.[RSM14b] as well as from the existing implementations to capture recent evolution of the system. The data flow of the coupling workflow has then been analyzed with respect to its potential for parallelization. We found that all stages of the workflow can be parallelized, most of them even with respect to multiple input dimensions. Then typical usage scenarios of the MapReduce paradigm and possible pitfalls that can prevent efficient use of this technology have been presented. Based on these findings, we described how the data aggregation and Octave parts of the workflow do indeed lend themselves very well to a MapReduce-based implementation. We also concluded that the first, PANDAS-based part is not suited for a MapReduce-based parallelization. We therefore limited our scope to the second part of the workflow and developed a detailed concept (see again Figure 4.2 for the resulting data and control flow). We also devised a concept of how MapReduce functionality can be exposed to workflows through a Web Service Interface (WSI).

In Section 5 we developed the aforementioned concept into an executable, Java-based prototype of the workflow. The prototype uses mapper and reducer scripts written in Python. Part of the prototype is also a simple implementation of the MapReduce Web Service (which we call MapReduce-WSI). This Web Service is able to import data from a RDBMS to HDFS, run MapReduce jobs (using either pre-compiled JAR archive or Streaming Mode scripts) and export the results while providing basic isolation between concurrent users of the service. In addition to the MapReduce prototype we developed a so-called baseline prototype, which represents a manual approach to parallelizing the simulation workflow across the cluster.

In Section 6 we evaluated the performance of the MapReduce and the baseline prototype. We varied both the size of the input data and the level of parallelism within the cluster. We showed that both prototypes scale to our expectations with respect to the input data size. We also confirmed our hypothesis that the MapReduce prototype has some overhead for scheduling and

setting up jobs on the cluster. However, the MapReduce prototype outperforms the baseline prototype with our largest test data set. Furthermore we observed some additional effects for which we presented theories. Larger data sets and more measurements would be required to draw more conclusions.

Lastly in Section 7, we showed that our Java-based MapReduce prototype can be easily converted into an equivalent and very straightforward workflow written in WS-BPEL. With respect to the initial goal, this shows that the integration of Big Data concepts into workflow technology is possible. The resulting workflow posesses the ability to scale to larger data sizes without any changes to the workflow itself. Also, aspects of monitoring and scheduling become available “for free” as natural part of the Hadoop ecosystem. With more classic approaches, such an infrastructure needs to be engineered, possibly cluttering the workflow itself.

While this is all very motivating, work on this thesis has also outlined some disadvantages:

- As they use the MapReduce-WSI from their workflows, workflow designers still need to write MapReduce scripts. This can be inconvenient since MapReduce design is oftentimes othogonal to classic, relational understanding of data. SQL queries are easier to write,to maintain and they are significantly more concise. Sometimes the structure of an algorithm can necessitate using multiple levels of mappers and reducers for relatively simple tasks. Also there is currently no other way of bundling dependencies and configuration than to embed them into the source code of the mapper and reducer scripts.
- The initial effort required to setup a functioning Hadoop cluster is high. Also, Hadoop is a complex system in which small configuration tweaks can have a huge impact on the performance of e.g, MapReduce or HDFS. Optimizing the configuration requires expert knowledge of a multi-layered stack of services as well as significant systems knowledge. Thus, using MapReduce-WSI or MapReduce in general will rarely be fully efficient with respect to computational resources. This has become clear in the *Straggler* effect we observed during measurements.

8.1 Future Work

With respect to the bone simulation, there is a variety of aspects to be further explored. The data aggregation and Octace part of the bone simulation has been successfully parallelized, however the PANDAS-based FEM simulation is still sequential even though it allows for parallelism at the granularity of a Motion Sequence. The MapReduce paradigm does not easily map to calculations that depend on complex Web Services which themselves require complex data provisioning and have side effects. A different parallelization concept will have to be developed to cover these cases. However, using MySQL Cluster as a scalable in-memory data store is promising since it would work directly with the existing PANDAS Web Service. Using MySQL

Cluster or similar technology, final results can be persisted and more intermediate data be kept in memory only.

An important part of this work has been to evaluate the flexibility of the resulting workflow prototypes. While the simplicity of the MapReduce prototype implementation is convincing, five major directions remain open for investigation which will be briefly covered in the following sections.

8.1.1 Alternatives to MapReduce: PACT

The disadvantages we noted are related to MapReduce as a paradigm and possibly Hadoop as a platform. Therefore other toolchains supporting large-scale parallel computations should be considered as a potential alternative. Based on our experience parallelizing the bone simulation, we can define two basic traits that are desirable in such a system:

- Ability to execute user-defined functions, i.e., run arbitrary simulation code on the cluster. A pure query system is not sufficient.
- The system can formulate chains of computations in a single batch as opposed to having to use multiple sequential jobs, as it is the case with MapReduce.

Possible candidates within the Apache Hadoop ecosystem include *Apache Hive* (which supports embedding user-defined mapper and reducer functions inside its SQL-like query language) and *Apache Spark*. Both have been briefly presented in Section 2.4.5.

The *PACT* framework has been proposed by Alexandrov et al. and Battré et al. in 2010 [AHM+10][BEH+10]. It further abstracts the MapReduce model into so-called *Parallelization Contracts* (i.e., PACTs). A Parallelization Contract is a second-order function that takes an user-defined function and a set of input data sets to perform a parallel operation, applying the user-defined function to subsets of the inputs. The *Map* and *Reduce* second-order functions in the MapReduce programing model are examples of PACTs. However, the system is extensible and more second-order functions can be added.

A *PACT* program is created by constructing an flow graph of separate PACTs. Such a flow graph is then processed by the *PACT Compiler*, which produces an optimized Directed Acyclic Graph (DAG) representation. This DAG is then executed across the cluster by the *Nephele Execution Engine*, which handles all runtime aspects [WK09].

To conclude, *PACT* is naturally taylored towards expressing complex data flows and therefore matches the above traits very well. It may be able to overcome some of the weaknesses of the current MapReduce-based approach and should thus be further evaluated.

8.1.2 Scaling to Larger Data Sets

Due to technical constraints the simulation has not been scaled to input data volumes larger than 30 GiB. It might be interesting to scale further, up into the Terabyte range. Using MySQL Cluster as an in-memory store may become infeasible at this point and a multi-tier storage solution may be required, the complexity of which will have to be hidden from the workflow in order to keep things simple. Also, this would further confirm the hypothesis we evaluated in Section 6 and help address some of our open theories.

8.1.3 Holistic Parallelization of the Bone Simulation

The entire simulation is designed only with respect to a single bone. It might be interesting to learn how the workflow can be adapted or re-use to simulate an entire skeleton consisting of hundreds or at least douzens of bones. Possible approaches include i) extending the simulation to include a Bone ID or ii) running a top-level simulation which then launches nested instances of the current simulation (one for each bone), possibly exchanging boundary information. While the second approach seems far cleaner from the perspective of a workflow designer, it may be significantly less efficient. A more detailed look into scheduling of jobs on computing clusters and possibly alternatives to MapReduce will be required.

8.1.4 Further Integration with SIMPL

To ease integration, a more generic way of accessing large-scale computing systems (of which Hadoop is only one, yet very popular representative) could be integrated into the SIMPL framework by Reimann et al.[RRS+11] which has been briefly described in Section 2.2.2. The framework provides abstract patterns for data access and transformation which decouple scientists and workflow designers from the details of how parallelization is achieved. When patterns are transformed into executable workflow fragments, use of Big Data concepts could be substituted based on heuristics. For example the complexity of moving data between a RDBMS and a NoSQL storage systems (such as HBase or raw HDFS) could be hidden entirely.

8.1.5 Other Simulation Models

While the results of this discussion allow for the conclusion that to introduce Big Data concepts into scientific workflow technology is worthwhile further pursuing, more research is required. This thesis has been focused on a very specific example of a simulation workflow, different case studies will be required to further generalize the results.

A Appendix A - WSDL for MapReduce-WSI

This is the WSDL diagram (generated using Eclipse's WSDL editor) to the MapReduce-WSI Web Service presented in Section 5.1. The full implementation can be found on Github[1].

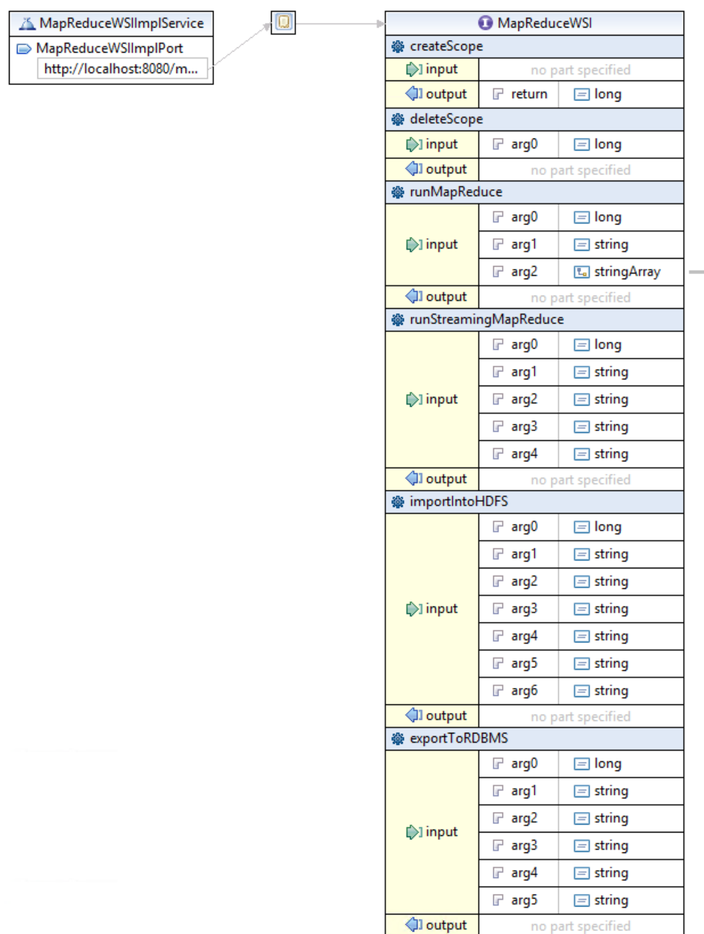


Figure A.1: WSDL Interface Description for MapReduce-WSI

[1] https://github.com/acgessler/mapreduce-wsi

B Appendix B - Code to Run and Measure the MapReduce prototype

```
package de.uni_stuttgart.ipvs_as.eval.mapreduce;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import de.uni_stuttgart.ipvs_as.MapReduceWSI;
import de.uni_stuttgart.ipvs_as.MapReduceWSIException;

// Simulation runner for the MapReduce prototype
public class Runner {

	// MapReduce-WSI binding
	public static final String WSDL_PATH = "http://localhost:8080/mapreduce-wsi/mapreduce?wsdl";
	public static final String SERVICE_SCOPE = "http://ipvs_as.uni_stuttgart.de/";
	public static final String SERVICE_NAME = "MapReduceWSIImplService";

	// Database name determines which data size the simulation runs on
	public static final String DB_NAME = "acg_eval_large";

	// This URI is accessed from within the cluster so localhost does not work
	public static String DB_URI = "jdbc:mysql://acg-mysql-1-1:3306/" + DB_NAME;

	public static final String DB_USER = "root";
	public static final String DB_PW = "root";

	public static final String DB_INPUT_TABLE_NAME = "mapreduce4sum_input";
	public static final String DB_OUTPUT_TABLE_NAME = "octave_output";

	public static final String HDFS_MR0_INPUT_NAME = "mr_input0";
	public static final String HDFS_MR0_OUTPUT_NAME = "mr_intermediate_0";
	public static final String HDFS_MR1_INPUT_NAME = "mr_intermediate_0";
```

```
    public static final String HDFS_MR1_OUTPUT_NAME = "mr_output1";

    public static final int ITERATIONS = 3;

    public static final String SQOOP_PARTITION_COLUMN =
          DB_INPUT_TABLE_NAME + ".elementid";

    public static final String SQOOP_IMPORT_QUERY =
          "SELECT motionid, elementid, gaussid, timestep, name, value " +
                "FROM " + DB_INPUT_TABLE_NAME +
          " WHERE (name = \"NS0\" OR name = \"SIG_V\" OR name = \"CNUF\")";

    public static final String DB_OUTPUT_SCHEMA = "CREATE TABLE " +
          DB_OUTPUT_TABLE_NAME + "(\n" +
          "-- ID of the FE Element Point\n" +
          "elementid INT,\n" +
          "-- ID of the Gaussian Point\n" +
          "gaussid INT,\n" +
          "\n" +
          "-- Output variables,\n" +
          "NSTS FLOAT,\n" +
          "HCF FLOAT\n" +
    ") ENGINE=NDBCLUSTER;";


    public static final String MR0_MAPPER_FILE = "aggregate_mapper.py";
    public static final String MR0_REDUCER_FILE = "aggregate_reducer.py";
    public static final String MR1_MAPPER_FILE = "octave_mapper.py";
    public static final String MR1_REDUCER_FILE = "octave_reducer.py";

    public static final String OCTAVE_CALC_FILE = "octave_calc.m";

    private static String readFile(String path) throws IOException {
          return new Scanner(new File(path)).useDelimiter("\\Z")
                .next().replace("\r\n", "\n");
    }

    private static Connection openDBConnection() throws SQLException {
          return DriverManager.getConnection(DB_URI, DB_USER, DB_PW);
    }

    private static void createOutputTable() throws SQLException {
          final Connection conn = openDBConnection();
          final Statement stat = conn.createStatement();

          // Clear output table and re-create it using the newest schema
          stat.execute(String.format("DROP TABLE IF EXISTS %s;", DB_OUTPUT_TABLE_NAME));

          stat.execute(DB_OUTPUT_SCHEMA);
          stat.close();
          conn.close();
```

```java
}

public static void main(String[] arguments) throws IOException, SQLException {
	// Dynamically connect to the MapReduce-WSI service instance
	URL url = new URL(WSDL_PATH);
	QName qname = new QName(SERVICE_SCOPE, SERVICE_NAME);
	Service service = Service.create(url, qname);
	MapReduceWSI port = service.getPort(MapReduceWSI.class);

	// Read mapper/reducer scripts
	final String mr0Mapper = readFile(MR0_MAPPER_FILE);
	final String mr0Reducer = readFile(MR0_REDUCER_FILE);
	final String mr1Mapper = readFile(MR1_MAPPER_FILE);
	final String mr1Reducer = readFile(MR1_REDUCER_FILE).replace(
			"OCTAVE_SOURCE_CODE_INSERT",
			// String escaping done right - twice!
			"\"" + readFile(OCTAVE_CALC_FILE)
				.replace("\\n", "\\\\n")
				.replace("\n", "\\n")
				.replace("\"", "\\\"")
				.replace("\'", "\\\'") +
				"\"");

	long delta0 = 0;
	long delta1 = 0;
	for (int i = 0; i < ITERATIONS; ++i) {
		System.out.println("Iteration: " + i);
		// Create output database table first
		createOutputTable();

		long time0 = System.nanoTime();

		long scope;
		try {
			// Create a new MapReduce-WSI scope
			scope = port.createScope();

			// Export data to HDFS
			port.importIntoHDFS(scope, DB_URI, DB_USER, DB_PW,
					SQOOP_IMPORT_QUERY,
					SQOOP_PARTITION_COLUMN,
					HDFS_MR0_INPUT_NAME);

			// Run MR0: data aggregation
			port.runStreamingMapReduce(scope, mr0Mapper, mr0Reducer,
					HDFS_MR0_INPUT_NAME,
					HDFS_MR0_OUTPUT_NAME);

			long time1 = System.nanoTime();

			// Run MR1: octave
```

```
                        port.runStreamingMapReduce(scope, mr1Mapper, mr1Reducer,
                                    HDFS_MR1_INPUT_NAME,
                                    HDFS_MR1_OUTPUT_NAME);

                        // Re-import results into SQL
                        port.exportToRDBMS(scope, DB_URI, DB_USER, DB_PW,
                                    DB_OUTPUT_TABLE_NAME, HDFS_MR1_OUTPUT_NAME);

                        // Free the HDFS scope
                        port.deleteScope(scope);

                        long time2 = System.nanoTime();
                        delta0 += (time1 - time0);
                        delta1 += (time2 - time1);
                  } catch (MapReduceWSIException e) {
                        e.printStackTrace();
                  }
            }

            delta0 /= ITERATIONS;
            delta1 /= ITERATIONS;
            final double scaleToSeconds = 1 / 1000000000.0;
            System.out.println("aggregation_time: " + delta0 * scaleToSeconds);
            System.out.println("octave_time: " + delta1 * scaleToSeconds);
            System.out.println("total_time: " + (delta0 + delta1) * scaleToSeconds);
      }
}
```

Listing B.1: MapReduce Prototype Runner

C Appendix C - Code to Generate Artificial Test Data

This is to illustrate how the test database containing artificial pandas output data is generated. The code as given below works for MySQL, minor adjustments to SQL syntax are needed for use with PostgreSQL. To control the size of the database generated, the *COUNT_xxx* constants can be tweaked.

```
package de.uni_stuttgart.ipvs_as.eval;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Locale;

public class PandasDummyDataGen {

      public static final String DB_USER = "root";
      public static final String DB_PW = "root";
      public static final String DB_NAME = "acg_eval_small";

      public static final String DB_INPUT_TABLE_NAME = "mapreduce4sum_input";
      public static String DB_URI = "jdbc:mysql://acg-mysql-1-1:3306/" + DB_NAME;

      public static final String SCHEMA = "CREATE TABLE " + DB_INPUT_TABLE_NAME + "(\n" +
            "-- Unique ID of this Motion Sequence \n" +
            "motionid TINYINT,\n" +
            "-- ID of the FE Element Point\n" +
            "elementid SMALLINT,\n" +
            "-- ID of the Gaussian Point\n" +
            "gaussid TINYINT,\n" +
            "-- Name of the mathematical variable\n" +
            "name VARCHAR(5),\n" +
            "-- ID of the Timestep\n" +
            "timestep SMALLINT,\n" +
            "PRIMARY KEY(motionid, elementid, gaussid, name, timestep),\n" +
            "INDEX elemid_name_index(elementid, name),\n" +
            "-- Output variable value\n" +
            "value DOUBLE PRECISION\n" +
      ") ENGINE=NDBCLUSTER;";

      public static final int ROW_MEM_USAGE_ESTIMATE = 20;
```

```
// Variable names emitted by Pandas
public static final String[] VAR_NAMES = new String[] {
      // Pandas instance #1
      "NS0",
      "NS",
      "TSEG",
      "WFL1",
      "WFL2",
      "WFL3",
      "SIG11",
      "SIG22",
      "SIG33",
      "SIG12",
      "SIG23",
      "SIG_V",
      "WRKEL",
      "KF",
      "SIGS",
      "GROW",
      "NUT",
      "NSTS",
      // Pandas instance #2
      "WNU1",
      "WNU2",
      "WNU3",
      "CNUF"
};

// Tweakable count of motions, elements etc.
public static final int COUNT_MOTION = 1;
public static final int COUNT_TIMESTEP = 100;
public static final int COUNT_ELEMENT = 1000;
public static final int COUNT_GAUSS = 8;

public static final long COUNT_TOTAL = (long)COUNT_MOTION * COUNT_TIMESTEP *
      COUNT_ELEMENT * COUNT_GAUSS * VAR_NAMES.length;

// Lower bound for output table size in MiB, assuming no compression
public static final long TOTAL_MEM_ESTIMATE_MIB = ROW_MEM_USAGE_ESTIMATE *
      COUNT_TOTAL / (1024 * 1024);

// Number of threads to use for populating the database
public static final int CONCURRENCY = 25;

public Connection openDBConnection() throws SQLException {
      return DriverManager.getConnection(DB_URI, DB_USER, DB_PW);
}

// Utility to collect inserted tuples, once a threshold is reached,
// the query is submitted to the database. This avoids hitting
```

```
// the maximum package size accepted by the server and keeps the
// number of queries sent small.
private class BatchInserter {
      private Statement stat;
      private Connection conn;

      private StringBuilder query;
      private int count = 0;

      // Row threshold at which to submit a batch to the server
      public static final int THRESHOLD = 25000;

      public BatchInserter() throws SQLException {
            rewind();
      }

      public void add(String tuple) throws SQLException {
            if (count > 0) {
                  query.append(",");
            }

            query.append(tuple);

            if (++count > THRESHOLD) {
                  submit();
                  rewind();
            }
      }

      public void finish() throws SQLException {
            if (count > 0) {
                  submit();
            }
            conn.close();
            stat.close();
      }

      private void submit() throws SQLException {
            query.append(";");
            for (int i = 0; ; ++i) {
                  try {
                        stat.executeUpdate(query.toString());

                  } catch(SQLException ex) {
                        System.out.println("Mysql error, retrying. " + ex.toString());

                        conn.close();
                        stat.close();
                        conn = openDBConnection();
                        stat = conn.createStatement();
```

```
                        try {
                              Thread.sleep(1000 * Math.min(10, (i + 1)));
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        continue;
                  }
                  break;
            }
      }

      private void rewind() throws SQLException {
            conn = openDBConnection();
            stat = conn.createStatement();

            query = new StringBuilder();
            query.append("INSERT INTO ");
            query.append(DB_INPUT_TABLE_NAME);
            query.append("(motionid, elementid, gaussid, name, timestep, value) VALUES");
            count = 0;
      }
};

// Populate with data for a range of FE elements
private void populateDataPartition(int partitionId, int beginElem, int endElem)
      throws SQLException
{
      final BatchInserter inserter = new BatchInserter();

      final long percentStep = COUNT_TOTAL / (100 * CONCURRENCY);
      long valInserted = 0;
      for (int mid = 0; mid < COUNT_MOTION; ++mid) {
            for (int feid = beginElem; feid < endElem; ++feid) {
                  for (int gaussid = 0; gaussid < COUNT_GAUSS; ++gaussid) {
                        for (int tid = 0; tid < COUNT_TIMESTEP; ++tid) {
                              for (String name : VAR_NAMES) {

                                    inserter.add(String.format(Locale.US,
                                                "(%d, %d, %d, \'%s\', %d, %f)",
                                                mid, feid, gaussid, name, tid,
                                                Math.random()));

                                    if (valInserted++ % percentStep == 0) {
                                          final long pct = CONCURRENCY * 100
                                                      * valInserted / COUNT_TOTAL;
                                          final long estimate = CONCURRENCY
                                                      * TOTAL_MEM_ESTIMATE_MIB * valInserted
                                                      / COUNT_TOTAL;

                                          final String s = String.format(
```

```
                                                        "Thread #%d inserted %d rows, ~%d%%
                                                            of total, ~%d MiB",
                                                        partitionId,
                                                        valInserted, pct,
                                                        estimate);
                                            System.out.println(s);
                                            System.out.flush();
                                        }
                                    }
                                }
                            }
                        }
                    }
                    inserter.finish();
                }


        public void initDBContents() throws SQLException {
                final Connection conn = this.openDBConnection();
                final Statement stat = conn.createStatement();

                // Clear input table
                try {
                        stat.execute(String.format("DROP TABLE %s;", DB_INPUT_TABLE_NAME));
                } catch (SQLException e) {
                        // Fine for the first time.
                        e.printStackTrace();
                        System.out.println("This can be ignored if the generator is is run"
                                        + " for the first time or the DB has been cleared.");
                }

                // Set schema for input table
                // http://johanandersson.blogspot.de/2010/04/tuning-your-cluster-with-ndbinfo-71.html
                stat.execute("set ndb_table_no_logging=1;");
                stat.execute(SCHEMA);
                stat.execute("set ndb_table_no_logging=0;");

                // Populate the input table in larger batches using multiple threads
                final int partitionSize = COUNT_ELEMENT / CONCURRENCY;
                for (int partition = 0; partition < CONCURRENCY; ++partition) {
                        final int threadPartition = partition;
                        new Thread() {
                                public void run() {
                                        try {
                                                int beginElem = threadPartition * partitionSize;
                                                int endElem = beginElem + partitionSize;
                                                if (COUNT_ELEMENT - endElem < partitionSize) {
                                                        endElem = COUNT_ELEMENT;
                                                }

                                                populateDataPartition(threadPartition, beginElem, endElem);
```

```
                        } catch (SQLException e) {
                            e.printStackTrace();
                        }
                    }
                }.start();
            }

            stat.close();
            conn.close();
        }

        /**
         * @param args
         */
        public static void main(String[] args) {
            try {
                (new PandasDummyDataGen()).initDBContents();
            } catch (SQLException e) {
                System.out.println("Error, check SQL commands and DB settings");
                e.printStackTrace();
            }
        }
}
```

Listing C.1: Java Code to Populate the *pandas_output* Table

Bibliography

[AHM+10] A. Alexandrov, M. Heimel, V. Markl, D. Battré, F. Hueske, E. Nijkamp, S. Ewen, O. Kao, D. Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *Proc. VLDB Endow.*, 3(1-2):1625–1628, 2010. doi:10.14778/1920841.1921056. URL http://dx.doi.org/10.14778/1920841.1921056. (Cited on pages 10 and 89)

[Apa13] Apache Foundation. Hadoop Streaming, 2013. URL http://hadoop.apache.org/docs/r1.2.1/streaming.html. (Cited on page 18)

[BCH13] L. A. Barroso, J. Clidaras, U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition*. Morgan & Claypool Publishers, 2013. (Cited on pages 27 and 28)

[BEH+10] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 119–130. ACM, New York, NY, USA, 2010. doi:10.1145/1807128.1807148. URL http://doi.acm.org/10.1145/1807128.1807148. (Cited on page 89)

[Boh14] A. Bohrn. Pattern-basierte Definition der Datenbereitstellung für Simulationen zu Strukturänderungen in Knochen. Studienarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Anwendersoftware, 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2448&engl=0. (Cited on pages 14, 16, 36 and 37)

[CDE+12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pp. 251–264. USENIX Association, Berkeley, CA, USA, 2012. URL http://dl.acm.org/citation.cfm?id=2387880.2387905. (Cited on pages 22 and 23)

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008. doi:10.1145/

1365815.1365816. URL http://doi.acm.org/10.1145/1365815.1365816. (Cited on pages 17 and 22)

[CRP+10] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, N. Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. *SIGPLAN Not.*, 45(6):363–375, 2010. doi:10.1145/1809028.1806638. URL http://doi.acm.org/10.1145/1809028.1806638. (Cited on page 17)

[CSF06] J. Cardoso, A. P. Sheth, Frank Leymann (Foreword). *'Foreword', Semantic Web Services, Processes and Applications*. Springer, 2006. (Cited on page 11)

[DG08] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492. (Cited on pages 9, 16, 17, 18 and 21)

[Dor11] R. Dormien. *Service-Bus-Erweiterung um Pandas-basierte Simulationen in Workflows zu nutzen*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3127&engl=0. (Cited on page 16)

[GGL03] S. Ghemawat, H. Gobioff, S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43. ACM, New York, NY, USA, 2003. doi:10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450. (Cited on page 18)

[GNC+09] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009. doi:10.14778/1687553.1687568. URL http://dx.doi.org/10.14778/1687553.1687568. (Cited on page 21)

[GSK+11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, M. Reiter. Conventional Workflow Technology for Scientific Simulation. In X. Yang, L. Wang, W. Jie, editors, *Guide to e-Science*, Computer Communications and Networks, pp. 323–352. Springer London, 2011. doi:10.1007/978-0-85729-439-5_12. URL http://dx.doi.org/10.1007/978-0-85729-439-5_12. (Cited on page 12)

[Knu97] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. (Cited on page 52)

[KSR+11] R. Krause, D. Schittler, M. Reiter, S. Waldherr, F. Allgöwer, D. Karastoyanova, F. Leymann, B. Markert, W. Ehlers. Bone remodelling: A combined biomechanical and systems-biological challenge. *PAMM*, 11(1):99–100, 2011. doi:10.1002/

pamm.201110041. URL http://dx.doi.org/10.1002/pamm.201110041. (Cited on page 15)

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000. URL http://books.google.de/books?id=Xc8eAQAAIAAJ. (Cited on page 9)

[MK13] S. G. Mats Kindahl, Neha Kumari. MySQL Applier for Apache Hadoop. Presented as the MySQL Connect conference 2013 in San Francisco, CA, USA, 2013. (Cited on page 23)

[OAS07] OASIS Open Standards. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf. (Cited on pages 12 and 13)

[oct] GNU Octave Website. URL http://www.gnu.org/software/octave. (Cited on pages 9 and 15)

[pan] Pandas Framework Website. URL http://www.mechbau.uni-stuttgart.de/pandas/index.html. (Cited on pages 9 and 15)

[Rie14] V. Riempp. *Pattern-basierte Kopplung eines biomechanischen und eines systembiologischen Simulationsmodells (Study Thesis Nr. 2449)*. Study thesis, University of Stuttgart, Universitaetsstr. 38, D-70569 Stuttgart, 2014. (Cited on page 16)

[RRS+11] P. Reimann, M. Reiter, H. Schwarz, D. Karastoyanova, F. Leymann. SIMPL - A Framework for Accessing External Data in Simulation Workflows. In *BTW*, pp. 534–553. Kaiserslautern, Germany, 2011. (Cited on pages 3, 4, 14 and 90)

[RSM14a] P. Reimann, H. Schwarz, B. Mitschang. A Pattern Approach to Conquer the Data Complexity in Simulation Workflow Design. In R. M. et al., editor, *Proceedings of OnTheMove Federated Conferences and Workshops (OTM), 22nd International Conference on Cooperative Information Systems (CoopIS 2014 in Amantea, Italy)*, volume 8841 of *LNCS*, pp. 21–38. Springer Berlin Heidelberg, 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-76&engl=. (Cited on pages 9, 15 and 59)

[RSM14b] P. Reimann, H. Schwarz, B. Mitschang. Data Patterns to Alleviate the Design of Scientific Workflows Exemplified by a Bone Simulation. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, SSDBM '14, pp. 43:1–43:4. ACM, New York, NY, USA, 2014. doi:10.1145/2618243.2618279. URL http://doi.acm.org/10.1145/2618243.2618279. (Cited on pages 9, 16, 35 and 87)

[Str10] C. Strozzi. NoSQL A Relational Database Management System, 2010. URL http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. (Cited on page 22)

[TDGS06] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Cited on page 9)

[TSJ+10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE '10: Proceedings of the 26th International Conference on Data Engineering*, pp. 996–1005. IEEE, 2010. doi:10.1109/icde.2010.5447738. URL http://dx.doi.org/10.1109/icde.2010.5447738. (Cited on page 22)

[WK09] D. Warneke, O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pp. 8:1–8:10. ACM, New York, NY, USA, 2009. doi:10.1145/1646468.1646476. URL http://doi.acm.org/10.1145/1646468.1646476. (Cited on page 89)

[ZKJ+08] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 29–42. USENIX Association, Berkeley, CA, USA, 2008. URL http://dl.acm.org/citation.cfm?id=1855741.1855744. (Cited on page 21)

All links were last followed on December 4, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature