AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND BIOMEDICAL ENGINEERING

DEPARTMENT OF APPLIED COMPUTER SCIENCE

Master of Science Thesis

Przetwarzanie i analiza danych multimedialnych w środowisku rozproszonym
Processing and analisys of multimedia in distributed systems

Author:	*Konrad Malawski*
Degree programme:	*Informatyka*
Supervisor:	*Sebastian Ernst PhD*

Kraków, 2014

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

I would like to thank . . . for all the support given to me during the creation of this thesis.

Contents

Todo list

1. Introduction

This section will introduce the problem areas covered by this thesis, briefly describing a few of the the use-cases of distributed multimedia analysis systems. It then introduces the two use-cases that are explicitly targeted by an reference system implemented as part of this thesis, in order to benchmark the usability of existing technologies in the area. Lastly it briefly outlines each of the following chapters.

1.1. General problem area

opisac og
czym tu r
dlaczego
sie na roz
i hadoop

The primary goal of this work is to research how to efficiently work with humongous amounts of multimedia data in a distributed setting, and weather this approach is the tight one.

In order to guarantee that recommendations and measurements made during this research are applicable in the "real world", outside of laboratory environments, I have defined a series of problems (described in the next section) and implemented a system which is able to solve those problems as well as easily adapt to any new requirements benefiting from the use of parallel access to hundreds of gigabytes of reference video material.

1.2. Reference system - investigated use cases

As stated in the introduction section, in order to be able reliably verify criteria such as responsiveness, cost-efficiency, performance and of course scalability of distributed systems, there must be some reference problem and solution the measurements will be made on.

For the sake of this paper I propose a "video material analysis platform" from here on referred to as the "*Oculus*" system. In the following two sub sections I will explain the use cases (thus - requirements) this system will aim to solve, which while being a very interesting topic of research by itself, will provide me a platform to measure the usefulness of the selected distributed system building blocks used for it.

1.2.1. Near–duplicate detection

One of the simplest use cases in which this system might be used is *near–duplicate detection* of video files. Note the term "near–duplicate" here, as exact duplicates are not the biggest problem – and the designed system must also be able to identify ”almost identical” material.

It might be easier to imagine the bellow use-cases if we think of a system like *youtube.com* **??**, where vast amounts of content are uploaded *each second*. An example of why ”almost identical” material in this setting would be a movie trailer, which has just been released and many fans want to put it online on youtube, in order to share this trailer. It is very likely that they would add slight modifications, such as their own voice-over with comments, or resize the video for example. It is also fairly common that users apply malicious modifications to the video material in order to make 1:1 identification with copyrighted material harder - such modifications are typically ”mirroring” the video material, or slightly brightening every frame. The system proposed in this work identifies content properly even after such modifications have been applied to the source content.

1.2.2. Data extraction

Another, less copyright focused, goal of the presented system is to be able to extensively mine data directly from the uploaded video content. Here the canonical example would be a ”*TOP 10 Movies of All Time*” video, which obviously contains video material from at least 10 movies, usualy in the order of 10th, 9th ... until the 1st (best) movie of all time. If we would be able to match parts of each video to their corresponding reference materials, we would be able to get meta data about the now recognised movies and even mine out the data what is the best / worst movie of all time, even without it being written per se – only by looking at the frames in the video.

While talking about extracting data from movies one cannot skip extracting text from images which seems both one of the most valuable things we can extract as well as with existing open source solutions for text recognition should not be hard to enable, given all the previous work which will be required to fulfil the above use case.

1.3. Thesis structure

In section 1 I will describe the architecture of the system, and briefly go over how this architecture benefits future extension.

In section 2 I will focus in the technical challenges encountered and solved during implementing the reference system.

The last section

done
one with
) Oculus

2. Analysis of available technologies

As the core of this work will focus on analysing and benchmarking usage of popular distributed system stacks, it is only fair to begin with introducing the selected components from which the system consist.

This chapter should be treated as a brief introduction into the selected technologies, as very detailed explanations and and implementation details will be provided throughout chapters 3 through 4.

2.1. Apache Hadoop

Apache Hadoop is a suite of tools and libraries modeled after a number of Google's most fameous whitepapers concerning "Big Data", such as *Chubby* [**?**] (on which the *Google Distributed File System* [**?**] was built) and later papers like the ground breaking *Map Reduce* [**?**] whitepaper concerning parallelisation of computation over massive amounts of data. The re-implementation of these whitepapers which has become known as Hadoop was originaly an implementation used by Yahoo [**?**] internally, and then released to the general public in late 2007 under the Apache Free Software License.

The general use-case of Hadoop based system revolves around massively parallel computation over humongous amounts of data. Thanks to employing functional programming paradigms in multi-server environments Hadoop makes it possible, and simple, to distribute so called "Map Reduce Jobs" across thousands of servers which execute the given *map* (also known as "*transform*") and *reduce* (also known as "*fold*") functions in a paralle, distributed fashion. Complex computations, which can not be represented as single Map Reduce jobs, are often executed as a series of jobs, so called Job Pipelines. This method will be leveraged and explained in detail in Chapter **??**, together with the introduction of Scalding (see Section 2.2) a Domain Specific Language built specifically to ease building such pipelines.

The promise of Hadoop is practically linear scalability of Hadoop clusters when adding more resources to such cluster – these claims will be investigates in Chapter 4, where results of different cluster configurations will be compared. The computation model proposed by Hadoop will be examined and explained in detail in later sections of this paper, as it is the dominating model chosen for the implementation of the presented system.

2.2. Scalding & Cascading

Scalding is a Domain Specific Language implemented using the Scala [?] programming language developer at Twitter [?] for their internal needs, and then released under the GPL license. It is aimed at proving a more expressive and powerful language for writing Map Reduce Job definitions, which otherwise would be implemented in the Java [?], which would often result in very verbose and hard to understand code (especialy due to the verbosity of Hadoop's core APIs).

Scalding is a thin layer on top

Cascading is a framework built on top Apache Hadoop and enables map reduce authors to think in terms of high level abstractions, such as data "flows" and job "pipelines" (series of Map Reduce jobs executed in parallel or sequentially) which have been used extensively in this project.

During the work on this thesis several I have pushed several contributions to the Scalding open source project.

utions

hes in-

t scald-
cading

2.3. Apache HBase

HBase is a column-oriented database [?] designed by following the Google white paper on their "BigTable" datastore published in 2007. Column oriented storage of data, as opposed to row oriented (as most SQL databases), as huge advantages when many aggregations over only given columns are performed.

It was selected for this project because it's excellent random-access to data as well as being perfectly suited for sourcing Map Reduce tasks. HBase stores it Tables on the Hadoop Distributed File System, thus it scales similarly to it, which will be proven in a later section in Chapter 4.

2.4. Scala

Scala is a functional *and* object-oriented programming language designed by Martin Odersky [?] running on the Java Virtual Machine. I selected it as primary implementation language for this project (though other languages used include: ANSI C, Ruby and Bash) because of the compelling libraries for building distributed systems using it, such as *Akka* and *Scalding* (introduced in the sections 2.5 and **??**).

It's functional nature (making it similar to languages such as Lisp or Haskell) is very helpful when performing transform / aggregate operations over collections of data. It should be also noted that Hadoop itself was inspired by languages such as this, because the canonical names of the functions performing data transformation and aggregation in functional languages are: "map" and "reduce".

2.5. Akka

Akka is a library providing an Actor Model [**?**] based concurrency for Scala (and Java) applications. This model may be familiar to some as it has gained popularity thanks to Erlang [**?**] which implements the same concepts.

For the sake of this thesis, Akka has been used both in local (in-jvm) parallel execution as well as remote (across nodes) message passing mode, in order to balance the workload generated by actors across the entire cluster. This is explained in detail in Chapter .

where?

2.6. phash

PHash is short for „Perceptual Hash" and is a sort of hashing algorithm (primarily aimed for use with images), which retains enough information to be comparable with another has, yielding „how similar" these hashes are. The details and implementation of it have been explained by Christoph Zauner's [**?**].

This algorithm is used by the system to perform initial similarity analysis between images. The algorithm is publicly available, including sources (in C), and may be used in non-commercial applications.

As the goal of this thesis is not introducing such algorithm, but focusing on image analysis in distributed systems, I decided to use the provided implementation and focus on clustering and scaling problems.

is this ne

2.7. Chef

Because of how tedious setting up Hadoop clusters is I have early on during the implementation phase of the project decided that cluster provisioning and configuration must be fully automated. Opscode's Chef is a tool which enables preparing provisioning scripts in a very readable way and apply them to a given set of machines.

Using it as well as cloud providers such as Amazon's EC2, Google's ComputeEngine I was able to fully automate a cluster's deployment.

can i noti

Details about the implementation of these recipes are featured in Appendix A.

2.8. Other tools and technologies used

2.8.1. youtube-dl

Youtube-dl is a small library written in python and freely available under an Open Source license. It was used in order to make downloading source video files from youtube more efficient, as it is aware of multiple available video formats (high / low quality), and offers multiple options useful yet hard to implement for this project – including for example „preferring to download open source video formats”, which allowed me to avoid installing proprietary video codecs on the servers.

2.8.2. tesseract-ocr

Tesseract [?] is a text recognition library developed by Google and freely available to use (including language stems for most popular languages). This tool has been used in order to extract text from analysed images, providing even more data.

3. System and design

The system, from here to be referred to by the name ”*Oculus*”, is designed with an asynchronous as well as distributed aproach in mind. In order to achieve high asynchrononisity between obtaining new reference data, and running jobs such as ”*compare video1 with the reference database*”, the system was split into two primary components:

– **loader** – which is responsible for obtaining more and more reference material. It persists and initially processes the videos, as well as any related metadata. In a real system this reference data would be provided by partnering content providers, yet for this

– **analyser** – which is responsible for preparing and scheduling job pipelines for execution on top of the Hadoop cluster and reference databases.

To further illustrate the separate components and their interactions Figure 3.1 shows the different interactions within the system.

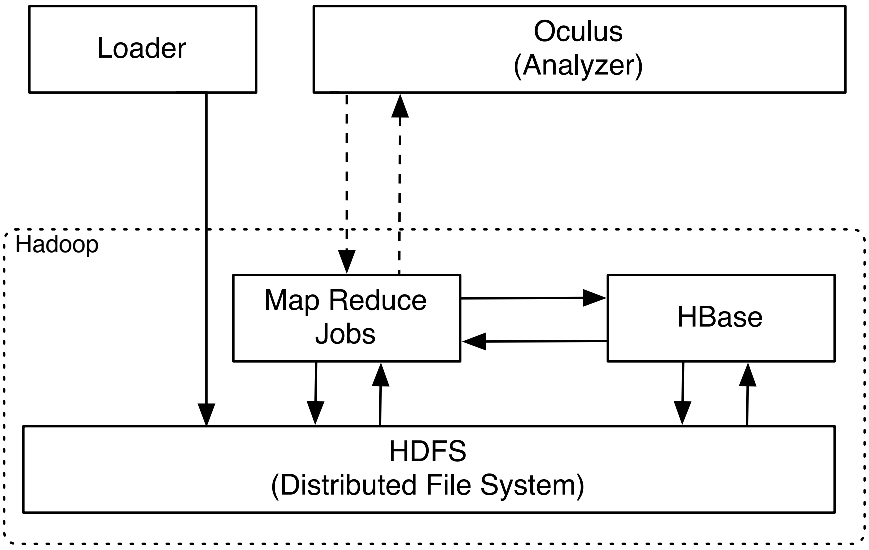

Figure 3.1: High level overview of the system’s architecture

3.1. Loader

The Loader component is responsible for obtaining as much as possible ”reference data”, by which I mean video material – from sites such as *youtube.com* or video hosting sites. Please note that for the sake of this thesis (and legal safety) the downloaded content was limited to movie trailers (which are freely available on-line) as well as series opening, ending sequences.

While I will refer to the Loader (as a system) in singular, it should be noted that in fact there are multiple instances of it running in the cluster. Thanks to the use of Akka’s [**?**] Actor Model abstractions (and *remoting* module [**?**]), in which the physical location of an Actor plays is of no importance – meaning, that the receiving Actor does not have to be on the same host as the sending Actor.

3.1.1. Types of Actors used in the system

The system consists of 4 types of Actors each of which has multiple instances which are spread out on many nodes in the cluster. Some tasks can only be sent to local Actors (any work requiring an already downloaded file), but messages related to crawling and initially downloading the video material can be spread throughout the cluster. I will now briefly describe the different Actor roles that exist in the system and then explain the interactions between then on an example.

– **YouTubeCrawlActor** – is capable of fetching and YouTube websites and generate Messages triggering either further crawling of ”related video sites” (`Crawl(siteId: String)`) or downloading of the currently accessed video (by sending a `Download(movieId)` message),

  receives:

  1 – `Crawl(siteId: String)` message

  sends:

  0 or n – `Crawl(siteId: String)` - where n is the number of ”related video” links found on the site. If crawling is turned off, no messages will be sent.

– **DownloadActor** – is responsible for downloading the movie from youtube in it’s original format (in the presence of many formats, the highest quality file will be downloaded). This Actor decides if a video is legal to download or not, because it also obtains the movie’s metadata – only trailers and opening sequences of series are downloaded during for the sake of this thesis.

– **ConversionActor** – is responsible for converting the downloaded video material into raw frame data (bitmaps).

  receives:

  `Convert(localVideoFile: java.util.File)` – This message must come from a local Actor, since the path refers to the local file system.

  sends:

`Upload(framesDirectory: java.util.File)` – when the finished converting to bitmaps, it will send and `Upload` message to one of the`HDFSUploadActors`, pointing to the directory where the output bitmaps have been written.

– **HDFSUploadActor** – is responsible for optimally storing the sequence of bitmaps in Hadoop. This includes converting a series of relatively small (around 2MB per frame) files into one Sequence File on HDFS. Sequence Files and the need for their use will be explained in detail in section 3.2.1.

receives:

`Upload(framesDirectory: java.util.File)` – pointing to a local directory where the bitmap files have been stored. This message must come from a local actor, since the path refers to the local file system.

3.1.2. Obtaining reference video material

In this subsection I will discuss the process of obtaining video material by the Loader subsystem, as well as explain which parts can be executed on different nodes of the cluster. The Figure 3.2 should help in understanding the basic workflow.

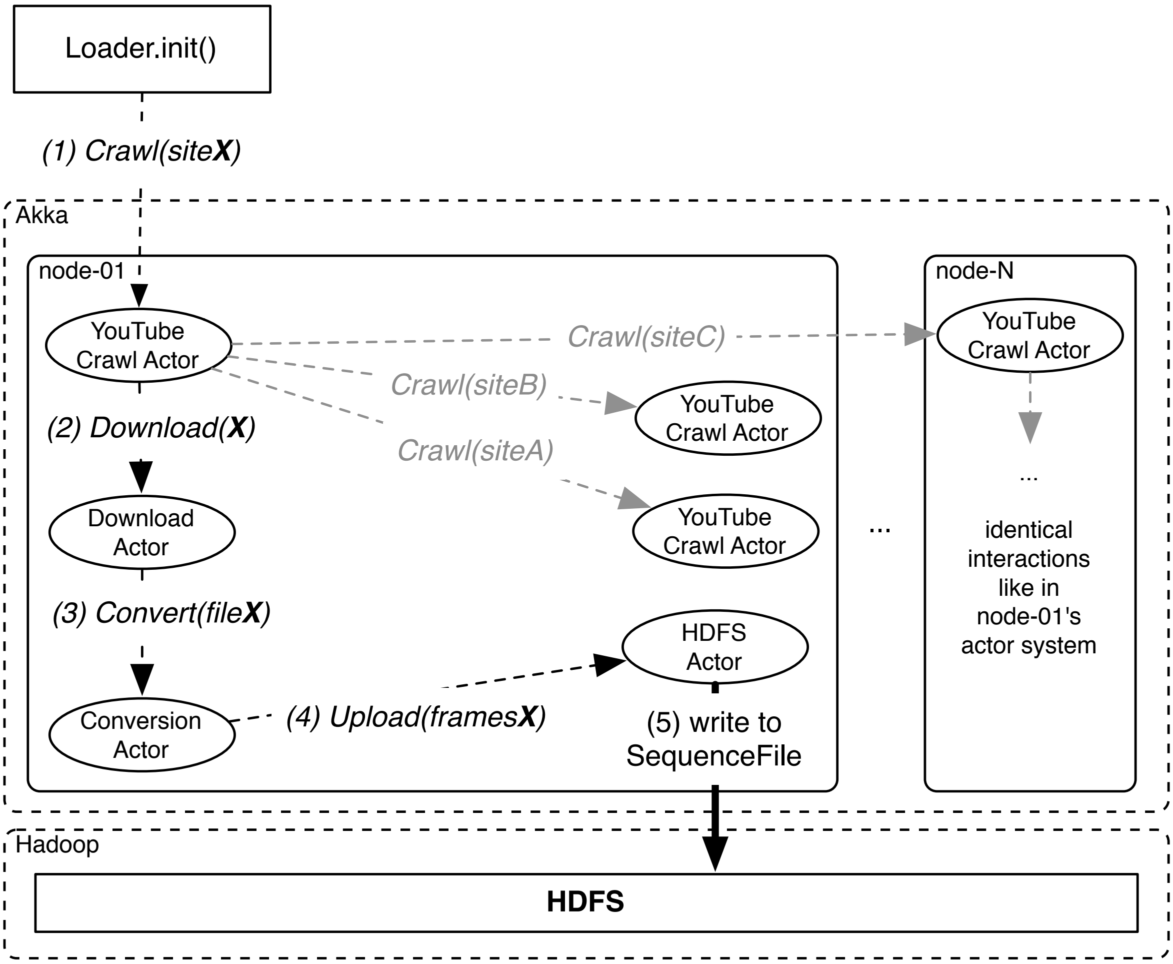

Figure 3.2: Overview of messages passed within the Loader's actor system. Greyed out messages are also sent, but are not on the critical path leading to obtaining material from *siteX* into HDFS.

Step 1 - *Crawl* messages

The initiating message for each flow within the Loader is a *Crawl(siteUrl)*, where siteUrl is a valid youtube video url. The receiving YouTubeCrawlActor will react to such message by fetching and extracting the related video site urls and will forward those using the same kind of *Crawl* messages. The second, yet most important, reaction is sending a *Download(movieId)* message to an instance of an DownloadActor – it can reside on any node in the cluster, which allows us to spread the down-link utilisation between different nodes in the cluster.

It is worth pointing out that the receivers of these messages can be remote Actors, that is, can be located on a different node in the cluster than the sender. In order to guarantee spreading of the load among the many actors within the system (across nodes in the cluster), I am using a strategy called ”Smallest Inbox Routing”. This technique uses a special ”Router Actor” which is responsible for a number of Routees (target Actors), and decides to deliver a message only to the Actor who has the smallest amount of messages ”not yet processed” (which are kept in an Queue called the ”Inbox”, hence the strategy's name).

Step 2 - *Download* messages

In the second step an *YouTubeDownloadActor* instance receives a message asking it to download a movie. It does so by invoking the native app *youtube-dl*, which is an open source program specialised in downloading movies from YouTube. Other than the video file (in an open source format) we also download a metadata description during this step, such metadata includes for example the date of publication, author, title and description of the movie. From this message including, all messages will be routed only to Actors local to the current node, because messages include *File* objects, pointing to locations on-disk.

The metadata is then used to determine if it can be used in the context of this work, as only movie trailers and and opening / ending sequences are downloaded into the Oculus system. If the content is OK to use, the Actor sends an *Convert(movieLocation)* message to an instance of *ConversionActor*.

Step 3 - *Convert* messages

The next step is executed by an instance of an *ConversionActor* recieving an *Convert(file)* message from another (local) Actor. The conversion phase will extract raw frame data from the incoming movie, and write those as plain bitmaps (not compressed) to files (one per each frame) into a specified target directory. The reason for not using a compressed lossless image format here is that all algorithms that the system will be dealing with later on are dealing with the raw image data, so we can avoid having to go over uncompression phases each time we will process a frame. Having that said, the storage format used for storing these files on HDFS provides build in compression (if enabeled), and it should be preferred in this case as it is transparent for the application, easing development of Map Reduce jobs in the Analyser system immensely.

The conversion from movie to raw bitmaps is performed by running an native application called `ffmpeg` [**?**] instance (an de facto standard tool for such media operations), by forking a process from within the Actor. The CPU usage of running this extraction process easily reaches 100% of the available resources, which is why the number of Conversion Actors per node is limited to only 1 per node, allowing ffmpeg to consume all available resources and finish extracting the data sooner. The actor will block until the process completes, and will then continue by sending an *Upload(bitmapDirectory)* message to one of the *HDFSUploadActors*.

Step 4 & 5 - *Upload* messages

The last step is an *HDFSUploadActor* recieving an *Upload(bitmapDirectory)* message which triggers it to connect to HDFS and start writing the bitmap data contained in the given directory to HDFS. The format of the generated data is as previously mentioned one file per frame of video, which averages around 2MB (depending on the movie resolution).

In this step the important part is that it does not write these files 1:1 onto HDFS, but instead writes into one file, using a hadoop specific storage format called ”Sequence File”, which allows for more efficient storage and latter retrieval of this data. Sequence Files, the need and benefits gained by using them as storage format for ”frame by frame” data will be discussed in Section 3.2.1.

This write terminates the operations performed on one movie by the Loader. All other operations will be performed by the Analyzer by running Map Reduce jobs on Sequence Files prepared in the above flow.

3.2. Analyser

The analyser component is responsible for orchestrating Map Reduce jobs and submitting them to the cluster. Results of jobs are written to either HBase or plain TSV (*Tab Separated Values*) Files. Figure 3.3 depicts the typical execution flow of an analysis process issued by Oculus.

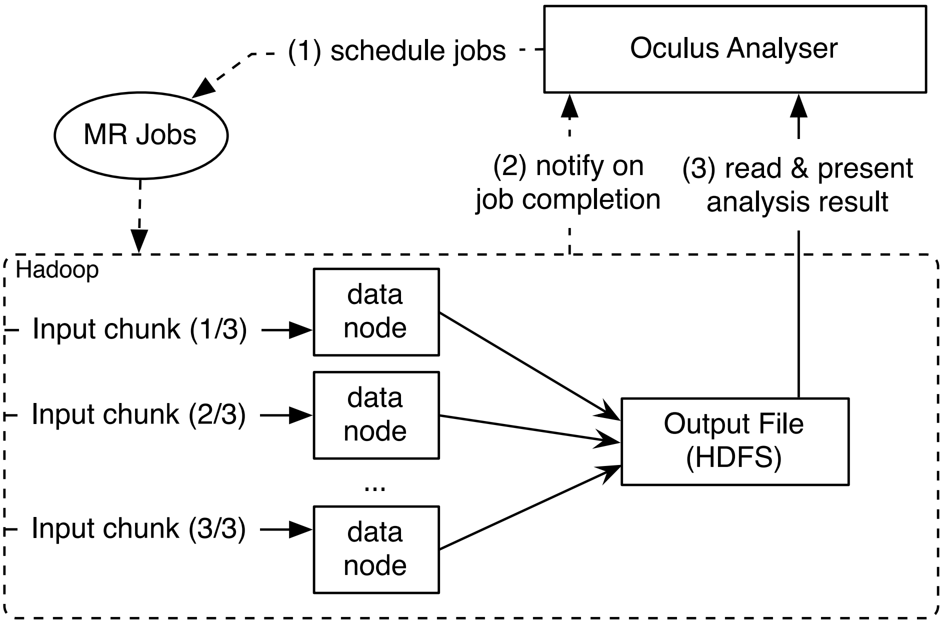

Figure 3.3: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes the *name-node* memory problems.

In step 1 the *job pipeline* is being prepared by the program by aggregating required metadata and preparing the job pipeline, which often consists of more than just one Map Reduce job – in fact, most analysis jobs performed by Oculus require at least 3 or more Map Reduce jobs to be issued to the cluster. It is important to note that some of these jobs may be dependent on another task's output and this cannot be run in parallel. On the other hand, if a job requires calculating all histograms for all frames of a movie as well as calculating something different for each of these frames – these jobs can be executed in parallel and will benefit from the large number of data nodes which can execute these jobs.

The 2nd step on Figure 3.3 is important because Oculus may react with launching another analysis job based on the notification that one pipeline has completed. This allows to keep different pipelines separate, and trigger them reactively when for example all it's dependencies have been computed in another pipeline.

For most applications though the 3rd step in a typical Oculus Job would be to read and present top N results to the issuer of the job, which for a question like ”Which movie is similar to this one?” would be the top N most similar movies (their names, identifiers as well as match percentage).

3.2.1. Frame-by-frame data and the HDFS ”small–files problem”

Most algorithms used in Oculus operate on a frame-by-frame basis, which means that it is most natural to store all data as ”data for frame 343 from movie XYZ”. This applies to everything from plain bitmap data of a frame to metrics such as histograms of colours of a given frame or other metadata like the extracted text content found in this frame.

Sadly this abstraction does not work nicely with Hadoop, it would cause the well–known ”small-files problem” which leads to *major* performance degradation of the Hadoop cluster is left undressed. In this section I will focus on explaining the problem and what steps have been taken to prevent it from manifesting in the presence of millions of ”by-frame” data points.

Hadoop uses so called ”blocks” as smallest atomic unit that can be used to move data between the cluster. The default block size is set to *64 megabytes* on most Hadoop distributions (including vanilla Apache Hadoop which this implementation is using).

This also means that if the DFS takes a write of one file (assuming the *replication factor* equals 1) it will use up one block. By itself this is not worrisome, because other than in traditional (local) file systems such as EXT3 for example, when we store N bytes in a block on HDFS, the the file system can still use block's unused space. Figure 3.4 shows the structure of a block storing only one frame of a movie.

The problem stemming from writing small files manifests not directly by impacting the used disk space, but in increasing memory usage in the clusters so called *name-node*. The name-node is responsible for acting as a lookup table for locating the blocks in the cluster. Since name-node has to keep 150KB of metadata for each block in the cluster, creating more blocks than we actually need quickly forces the name-node to use so much memory, that it may run into long garbage collection pauses, degrading the entire cluster's performance. To put precise numbers to this – if we would be able to store 500MB of data in an optimal way, storing them on HDFS would use 8 blocks – causing the name node to use approximately 1KB of metadata. On the other hand, storing this data in chunks of 2MB (for example by storing each frame of a movie, uncompressed) would use up 250 HDFS blocks, which results in

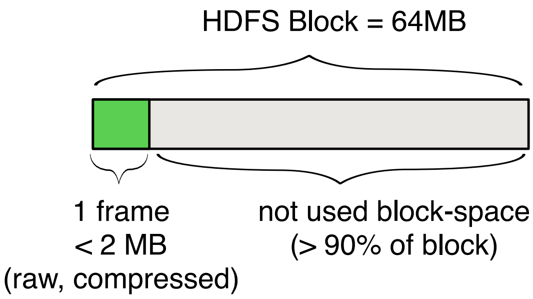

Figure 3.4: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes the *name-node* memory problems.

additional 36KB of memory used on the name-node, which is 4.5 times as much (28KB more) as with optimally storing the data! Since we are talking about hundreds of thousands of files, such waste causes a tremendous unneeded load on the name-node.

It should be also noted, that when running map-reduce jobs, Hadoop will by default start one map task for each block it's processing in the given Job. Spinning up a task is an expensive process, so this too is a cause for performance degradation, since having small files causes more *Map tasks* being issued for the same amount of actual data Hadoop will spend more time waiting for tasks to finish starting and collecting data from them than it would have to.

3.2.2. Defining Map Reduce Pipelines using Scalding and Cascading

DO THIS
AMPLES

TODO TODO THIS IS JUST SAMPLES

The primary language used for implementing all Oculus jobs, including Map Reduce jobs is Scala [?]

This is how an example job would look like:

Listing 3.1: Simplest Scalding job used in Oculus – each frame perceptual hashing

```
Tsv("input.tsv")
  .map(’line -> ’word) { line: String => line.split }
  .groupBy(’word) { _.count }
  .write(Tsv("output.tsv"))
```

Parallel execution and job ordering

Because a Scalding job (which effectively is an Cascading ”*Pipeline*”) can span multiple Map Reduce Job invocations, it is important to visualise how many actual Jobs will be submitted to the cluster and also if they can be run in parallel.

In order to visualise how jobs actually will be executed on the cluster Cascading provides a very important option allowing to print the resulting job graph using the widely accepted graph-description language DOT [**?**].

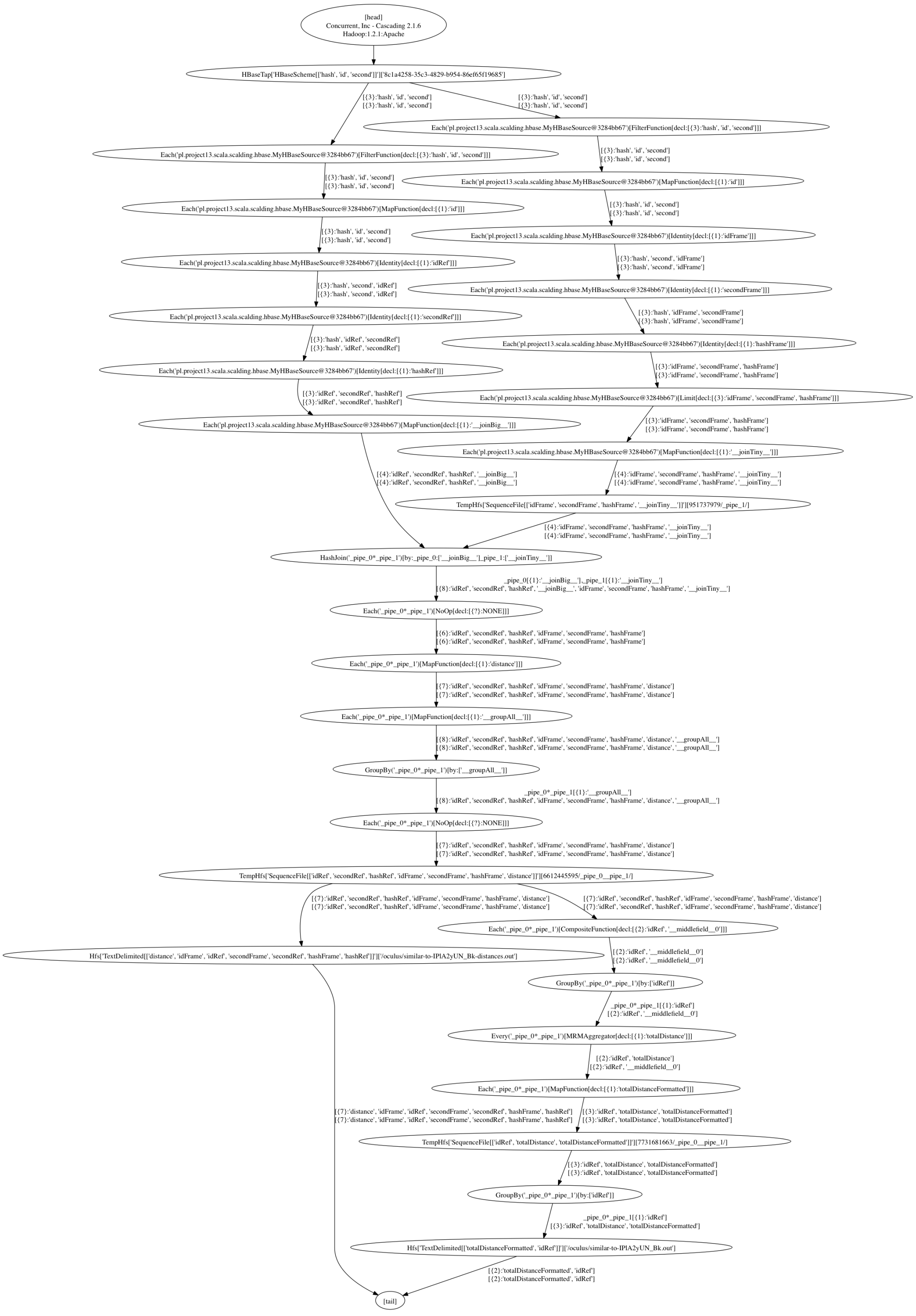
[head]
Concurrent, Inc - Cascading 2.1.6
Hadoop:1.2.1:Apache
HBaseTap['HBaseScheme[['hash', 'id', 'second']]']['8c1a4258-35c3-4829-b954-86ef65f19685']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[FilterFunction[decl:[{3}:'hash', 'id', 'second']]]
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[FilterFunction[decl:[{3}:'hash', 'id', 'second']]]
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[MapFunction[decl:[{1}:'id']]]
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[MapFunction[decl:[{1}:'id']]]
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
[{3}:'hash', 'id', 'second']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'idFrame']]]
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'idRef']]]
[{3}:'hash', 'second', 'idFrame']
[{3}:'hash', 'second', 'idFrame']
[{3}:'hash', 'second', 'idRef']
[{3}:'hash', 'second', 'idRef']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'secondFrame']]]
[{3}:'hash', 'idFrame', 'secondFrame']
[{3}:'hash', 'idFrame', 'secondFrame']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'secondRef']]]
[{3}:'hash', 'idRef', 'secondRef']
[{3}:'hash', 'idRef', 'secondRef']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'hashFrame']]]
[{3}:'idFrame', 'secondFrame', 'hashFrame']
[{3}:'idFrame', 'secondFrame', 'hashFrame']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Identity[decl:[{1}:'hashRef']]]
[{3}:'idRef', 'secondRef', 'hashRef']
[{3}:'idRef', 'secondRef', 'hashRef']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[Limit[decl:[{3}:'idFrame', 'secondFrame', 'hashFrame']]]
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[MapFunction[decl:[{1}:'__joinBig__']]]
[{3}:'idFrame', 'secondFrame', 'hashFrame']
[{3}:'idFrame', 'secondFrame', 'hashFrame']
Each('pl.project13.scala.scalding.hbase.MyHBaseSource@3284bb67')[MapFunction[decl:[{1}:'__joinTiny__']]]
[{4}:'idRef', 'secondRef', 'hashRef', '__joinBig__']
[{4}:'idRef', 'secondRef', 'hashRef', '__joinBig__']
[{4}:'idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']
[{4}:'idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']
TempHfs['SequenceFile[['idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']]'][951737979/_pipe_1/]
[{4}:'idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']
[{4}:'idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']
HashJoin('_pipe_0*_pipe_1')[by:_pipe_0:['__joinBig__']_pipe_1:['__joinTiny__']]
_pipe_0[{1}:'__joinBig__'],_pipe_1[{1}:'__joinTiny__']
[{8}:'idRef', 'secondRef', 'hashRef', '__joinBig__', 'idFrame', 'secondFrame', 'hashFrame', '__joinTiny__']
Each('_pipe_0*_pipe_1')[NoOp[decl:[{?}:NONE]]]
[{6}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame']
[{6}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame']
Each('_pipe_0*_pipe_1')[MapFunction[decl:[{1}:'distance']]]
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
Each('_pipe_0*_pipe_1')[MapFunction[decl:[{1}:'__groupAll__']]]
[{8}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance', '__groupAll__']
[{8}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance', '__groupAll__']
GroupBy('_pipe_0*_pipe_1')[by:['__groupAll__']]
_pipe_0*_pipe_1[{1}:'__groupAll__']
[{8}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance', '__groupAll__']
Each('_pipe_0*_pipe_1')[NoOp[decl:[{?}:NONE]]]
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
TempHfs['SequenceFile[['idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']]'][6612445595/_pipe_0__pipe_1/]
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
[{7}:'idRef', 'secondRef', 'hashRef', 'idFrame', 'secondFrame', 'hashFrame', 'distance']
Each('_pipe_0*_pipe_1')[CompositeFunction[decl:[{2}:'idRef', '__middlefield__0']]]
Hfs['TextDelimited[['distance', 'idFrame', 'idRef', 'secondFrame', 'secondRef', 'hashFrame', 'hashRef']]']['/oculus/similar-to-IPlA2yUN_Bk-distances.out']
[{2}:'idRef', '__middlefield__0']
[{2}:'idRef', '__middlefield__0']
GroupBy('_pipe_0*_pipe_1')[by:['idRef']]
_pipe_0*_pipe_1[{1}:'idRef']
[{2}:'idRef', '__middlefield__0']
Every('_pipe_0*_pipe_1')[MRMAggregator[decl:[{1}:'totalDistance']]]
[{2}:'idRef', 'totalDistance']
[{2}:'idRef', '__middlefield__0']
Each('_pipe_0*_pipe_1')[MapFunction[decl:[{1}:'totalDistanceFormatted']]]
[{7}:'distance', 'idFrame', 'idRef', 'secondFrame', 'secondRef', 'hashFrame', 'hashRef']
[{7}:'distance', 'idFrame', 'idRef', 'secondFrame', 'secondRef', 'hashFrame', 'hashRef']
[{3}:'idRef', 'totalDistance', 'totalDistanceFormatted']
[{3}:'idRef', 'totalDistance', 'totalDistanceFormatted']
TempHfs['SequenceFile[['idRef', 'totalDistance', 'totalDistanceFormatted']]'][7731681663/_pipe_0__pipe_1/]
[{3}:'idRef', 'totalDistance', 'totalDistanceFormatted']
[{3}:'idRef', 'totalDistance', 'totalDistanceFormatted']
GroupBy('_pipe_0*_pipe_1')[by:['idRef']]
_pipe_0*_pipe_1[{1}:'idRef']
[{3}:'idRef', 'totalDistance', 'totalDistanceFormatted']
Hfs['TextDelimited[['totalDistanceFormatted', 'idRef']]']['/oculus/similar-to-IPlA2yUN_Bk.out']
[{2}:'totalDistanceFormatted', 'idRef']
[{2}:'totalDistanceFormatted', 'idRef']
[tail]

Figure 3.5: This flow represents the most longest Pipeline preset in Oculus – finding which movies are similar to the current one, ordered by ranking. It is constructed from 5 Map Reduce jobs.

Figure 3.2.2 represents the ”Find similar movies to the given one” pipeline. Each circle represents an operation (such as emitting a tuple, grouping etc) and lines represent the data flowing through the Map Reduce jobs. It is also clearly visible that this pipeline consists of 5 map reduce jobs, where the 3rd job’s input depends on jobs 1 and 2, and only after job 3 has completed jobs 4 and 5 can be executed (in parallel).

Sometimes though it is not easy to determine from this diagram alone how many actual MR Jobs a pipeline has emitted. For this reason we can instruct Cascading to print a DOT file containing the ”steps”, which for the algorithm represented in Listing 3.2 Figure 3.2.2 would look like Figure 3.2.2.

Listing 3.2: Fragments of FindMostSimilarMoviesJob.dot

```
digraph G {
1 [label = "Hfs[’TextDelimited[[UNKNOWN] ->
           [’refHash’, ’frameHash’, ’distance’]]’][’out’]"];
2 [label = "Each(’_pipe_0*_pipe_1’)[NoOp[decl:[{?}:NONE]]]"];
3 [label = "GroupBy(’_pipe_0*_pipe_1’)[by:[’__groupAll__’]]"];
15 [label = "[tail]"];
# ...

4->3 [label = "[{4}:’refHash’, ’frameHash’, ’distance’, ’__groupAll__’]
            [{4}:’refHash’, ’frameHash’, ’distance’, ’__groupAll__’]"];
3->2 [label = "_pipe_0*_pipe_1[{1}:’__groupAll__’]
            [{4}:’refHash’, ’frameHash’, ’distance’, ’__groupAll__’]"];
1->15 [label = "[{3}:’refHash’, ’frameHash’, ’distance’]
            [{3}:’refHash’, ’frameHash’, ’distance’]"];
2->1 [label = "[{3}:’refHash’, ’frameHash’, ’distance’]
            [{3}:’refHash’, ’frameHash’, ’distance’]"];
# ...
}
```

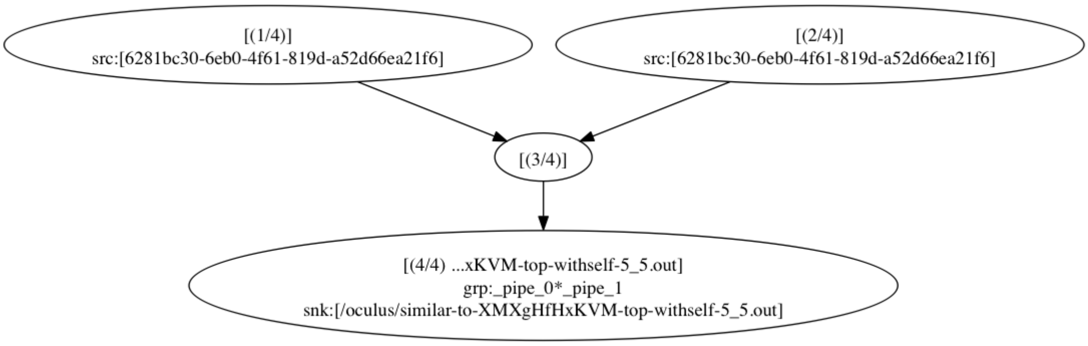
[(1/4)]
src:[6281bc30-6eb0-4f61-819d-a52d66ea21f6]
[(2/4)]
src:[6281bc30-6eb0-4f61-819d-a52d66ea21f6]
[(3/4)]
[(4/4) ...xKVM-top-withself-5_5.out]
grp:_pipe_0*_pipe_1
snk:[/oculus/similar-to-XMXgHfHxKVM-top-withself-5_5.out]

Figure 3.6: A graph representation of the Map Reduce jobs that have to be run in order to complete the pipeline.

the steps
wrong he

The graph represented on Figure 3.2.2 displays the same pipeline as Figure **??** but on a higher level – displaying only the order and dependencies of each Map Reduce job. Step `3/4` is easily identifiable as ”groupBy” here, although from this graph we are unable to determine on what field we’re grouping.

use I, not
not a blog

needs con
rewrite, s
examples,
code of th

Sequence Files

The solution applied in the implemented system to resolve the small files problem is based on a technique called ”Sequence Files”, which are a manually controlled layer of abstraction on top of HDFS blocks. There are multiple Sequence file formats accepted by the common utilities that Hadoop provides [**?**] but they all are *binary header-prefixed key-value formats*, as visualised Figure 3.7.

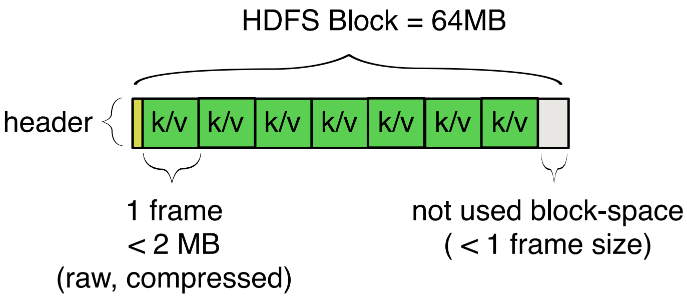

Figure 3.7: A SequenceFile allows storing of multiple small chunks of data in one HDFS Block.

Using Sequence Files resolves all previously described problems related to small files on top of HDFS. Files are no longer ”small”, at least in Hadoop’s perception, since access of frames of a movie is most often bound to access other frames of this movie we don’t suffer any drawbacks from such storage format.

Another solution that could have been applied here is the use of HBase and it’s key-value design instead of the explicit use of Sequence Files, yet this would not yield much performance nor storage benefits as HBase stores it’s Table data in a very similar format as Sequence Files. The one benefit from using HBase in order to avoid the small files problem would have been random access to any frame, not to ”frames of this movie”, but since I don’t have such access patterns and it would complicate the design of the system I decided to use Sequence Files instead.

4. Performance and scalability analysis

In this section I will analyse the devised system on such aspects as general performance, ability (and ease) of scaling the system to support larger amounts of data or to speed-up the processing times by adding more servers to the cluster.

These measurements will be made in the previously described system and backed by measurements

4.1. Scaling Hadoop

In this section I will investigate the impact of scaling the Hadoop cluster vertically (by adding more nodes) on processing times of movies.

The task used to measure will be the longest pipeline available in the oculus system - comparing a movie, percentage wise with all other movies in the database. This process touches a tremendous amount of data, and is also very parallelizable.

At first I measured the time to process the input movie ....

4.2. Scaling the Loader (actor system)

scale it to nodes...

5. Results and processing times

nothing sa
results ye

add some
ages so it

6. Conclusions

The applied technologies have indeed been very helpful, and proved to be very elastic for different kinds of jobs related to processing large amounts of data. I was also positively surprised with the ease of Scaling Hadoop infrastructure.

conclude

A. Automated cluster deployment

This chapter describes the automated tooling which has been used during the implementation of the reference system mentioned in this thesis in order to drastically increase turnaround time during development as well as cluster scaling.

Due to the complexities of maintaining possibly hundreds of virtual machines with similar (or even identical) configurations the time it would take to provision, configure and deplot applications on each new server in the cluster would render this process very slow and fiesable. Instead, tools and platforms have been applied to simplify and speed-up the turnaround time when adding new servers to the cluster.

In Section A the used cloud intrafstucture is introduced, along with a few examples of automating server provisioning using simple yet powerful command-line tools.

In Section A.1 Opscode Chef – the tool used to configure, as well as install dependencies and deploy applications is introduced.

A.1. Automated server provisioning – Google Compute Engine

In order to provision virtual machines for running the applicationc cluster Google's Compute Engine "Infrastructure as a Service" (also known under the acronym *IAAS*) was used.

Creating a new instance on GCE (*Google Compute Engine*) can be done via an admin console under `cloud.google.com` or using command line tooling (or plain JSON API calls). During this project the most used method was the command line API, as it is simple to prepare scripts for spinning up multiple VMs and combining this step with provisioning configuration to them using Chef (which will be explained in Section A.1. An example of how a new instance on GCE can be started is illustrated on Listing A.1.

Listing A.1: Creating new instance on GCE

```
gcutil --service_version="v1" --project="oculus-hadoop"
  addinstance "oculus-3"
  --machine_type="n1-standard-1"
  --zone="us-central1-a"
  --tags="hadoop,datanode"
  --disk="large-4,deviceName=large-4,mode=READ_WRITE"
  --network="default"
```

```
--external_ip_address="ephemeral"
--service_account_scopes="https://www.googleapis.com/auth/..."
--image="https://www.googleapis.com/.../images/debian-7-wheezy-v20140408"
--persistent_boot_disk="true"
--auto_delete_boot_disk="false"
```

Listing A.1 shows the current cluster's status.

s label

```
# gcutil listinstances

+---------------+---------------+---------+------------+------------+
| name          | zone          | status  | network-ip | pub-ip     |
+---------------+---------------+---------+------------+------------+
| oculus-1      | us-central1-a | RUNNING | 10.240.x.x | 23.236.x.x |
+---------------+---------------+---------+------------+------------+
| oculus-2      | us-central1-a | RUNNING | 10.240.x.x | 108.59.x.x |
+---------------+---------------+---------+------------+------------+
| oculus-master | us-central1-a | RUNNING | 10.240.x.x | 108.59.x.x |
+---------------+---------------+---------+------------+------------+
```

It it also possibe to invoke typical compute engine tasks using it's Chef (which is described in detail in Section A.1) plugins, so that an it's even easier to use and investigate the running cluster:

```
# knife google server list --gce-zone us-central1-a

name           type          public ip  disks        zone
oculus-1       n1-standard-1 23.x.x.x   d-1,large-4  us-central1-a
oculus-2       n1-standard-1 23.x.x.x   d-2,large-1  us-central1-a
oculus-master  n1-standard-1 23.x.x.x   m-0,large-3  us-central1-a
```

A.2. Automated configuration and deployment – Opscode Chef

Chef is a tool which enables to easily manage configuration and deployment of services and apps across cloud infrastructure. It consists of a set of tools using which one can describe a servers configurational requirements, such as what services it should have installed. It provides multiple ways to execute the provisioning step yet for the sake of this thesis the simplest ”solo” mode was used.

When using Chef in solo mode, one prepares a specific ”run$_{l}ist''thatconsistsofnamesofcookbooks(whicharesimplyaseriesof''stepstoexecute''inordertoprovisionsometh$

Listing A.2: Preparing and Cooking a server with in order to prepare it for becoming a Hadoop data-node

```
# knife solo prepare kmalawski@108.59.81.222 nodes/data-node.json
...
(Reading database ... 42465 files and directories currently installed.)
Preparing to replace chef 11.8.2-1.debian.6.0.5 (using .../chef_11.12.2-1_amd64.deb) ...
Unpacking replacement chef ...
Setting up chef (11.12.2-1) ...

# knife solo cook kmalawski@108.59.81.222 nodes/data-node.json
Uploading the kitchen...
```

finish

Hadoop's filesystem must be formated before put into use. This is achieved by issuing the `-format` command to the namenode:

```
kmalawski@oculus-master > hadoop namenode -format
```

It is worth pointing out that a "format" takes place only on the namenode, it does not actually touch the datab stored on the datanodes, but instead it deleted the data stored on the namenode. The Namenode, as explained previously, stores all metadata about where a file is located, thus, cleaning it's data makes the files stores in HDFS un-usable, since we don't know "where a file's chunks are stored".