Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

Przetwarzanie i analiza danych multimedialnych w środowisku rozproszonym

Processing and analisys of multimedia in distributed systems

Autor: Konrad Malawski

Kierunek studiów: Informatyka

Opiekun pracy: Dr Sebastian Ernst

Kraków, 2014

Oświadczam, świadoma odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałam osobiście i samodzielnie, i nie korzystałam ze źródeł innych niż wymienione w pracy.

. . . . . .

PODPIS

I would like to thank Dr Ernst, my friends and wife for 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. Goals of this thesis

The *primary goal* of this work is to research how to efficiently work with humongous amounts of multimedia data in a distributed setting. The results of this thesis include best practices and recommendations towards dealing with big-data applications, which have been aquired and tested while implementing a real system leveraging these lessons.

As will be described in more detail in Section 1.3, a large part of the work that was performed during this thesis has been implementing a reference system on which stress and scalability tests would then be performed, as in order to research scalability problems a sufficiently sophisticated reference system was needed to benchmark scaling methods on. The system itself will be dealing with simple image processing algorithms and near-duplcicate detection of movies, or scenes within movies.

For the sake of this thesis, the focus will be on video material, of which the amounts of freely and legally available materials are significant – esp. thanks to many materials licensed under the Creative Commons family of licenses [Com01].

1.2. General problem area

Image analysis over huge amounts of data is common place in todays world, where everything is recorded, published, possibly modified and distributed again, all this using digital media and digital storage formats. Starting from simple movies of cats uploaded to public video hosting services all the way through to sophisticated urban area monitoring services – everything can be, and is, recorded – yielding previously unbelievable amounts of digital multimedia data.

All this leads to the challanging task of efficiently handling this data. Tasks such as de-duplicating, searching, categorizing or extracting features (e.g. text) are now even more challanging and exciting than ever before. Together with the huge amounts of data these systems have to deal with, this new range of applications poses a significant challange and interesting opportunity to develop new kinds of paradigms as well as algorithms, geared specifinally towards dealing with big data and distributed computations.

1.3. Reference system – investigated use cases

In order to guarantee that recommendations and measurements made during this research are applicable in the ”real world”, outside of experimental environments, a set of problems has been defined and a ”reference system” has been implemented in order to solve these.

The system, from here on sometimes refered to as *”Oculus – the video material analysis platform”*, will be tasked at processing huge amounts of video data. The videos to be fed into the system will be scraped from publicly available video hosting websites, such as YouTube. It should be also made clear that videos imported into the system are all public domain or creative commons licensed material, so that even accidental copyright infrigement can be avoided.

The primary goal for this application is to expose and highlight challanges faced by application developers during the design, implementation and deployment phases of such applications. Using it as a point of reference, as well as test system, the problems given to the system (described in Sections 1.3.1 and 1.3.2) will be solved. Issues encountered during the implementation of that reference system will provide crucial hands–on expirience required in order to provide recommendations and best practices about building such systems – these will be captured in Chapter 5.

The next sections will expand on the tasks presented to the reference system.

1.3.1. Near–duplicate detection

One of the simplest use cases in which this system should be used is *near–duplicate detection* of video files. The term ”near–duplicate” is used in order to highlight the possibility (and anticipation of) distorted data. The system must be capable of identifying videos of slightly lower or higher quality than the reference material as the same movie. This use case is very near to what YouTube’s [Goo] internal copyright protection mechanisms are implementing – thus is is a valid as well as real–life usage scenario.

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 implemented as part of this thesis identifies content properly even after such malicious modifications have been applied to video materials.

1.3.2. Scene positioning

The problem of scene positioning can be explained as trying to locate ”when” during a full movie a given scene appears.

One might imagine a scenario in which a friend shows us a funny video from some series, available on-line. The snippet is only 30 seconds long – long enough to get the joke, but not long enough to figure out just based on this video from which episode, season or even from which show (if the scene movie was not properly titled) this scene comes from. A user might be intrigued by this scene and willing to pay a the content owner for viewing the entire series.

Instead of putting ourselfes in the position of taking down such copyrighted material, a system could detect from which exact show, season and episode the scene originates from and offer the user an option to, for example: ”See the whole episode at HubbleTube!”. The fictionary service HubbleTube could be a paid service, yet thanks to the convinience of directly linking to the exact content the user wants to see – the user is more willing to continue and pay to see the show. This way the content owner also profitsm, without having to take down any of his copyrighted content – instead it was used as crowd-sourced advertisement vector.

In order to enable use-cases like this, scene detection and positioning must be implemented within the reference system. A detailed analysis of this problem and results achieved by *Oculus* will be shown Section 4.2.

1.4. Thesis structure

Chapter 1 seved as broad overview and introduction into the problem area of this thesis. It also introduces the need for an reference implementation on top of which recommendations will be made in latter chapters of the thesis. Lastly, a number of goals are given to the reference system.

Chapter 2 focuses on describing the available and selected technologies used in this project. It also explains the choice of tools, as well as briefly introduces paradigms implemented by them, such as distributed file systems or the concept of *Map Reduce* [DG04] based applications.

Chapter 3 describes the overall design choices as well as flows of data throughout the system. It covers two applications which together form the ”reference system” named Oculus, on which experiments as well as tuning will be performed in the following chapters.

Chapter 4 provides examples and results of using the system in scenarios which have been given to it as part of it’s goals (in Chapter 1). A brief discussion on applied and possible optimizations closes this chapter.

Chapter 5 focuses on performance measurements as well as tuning techniques applied and recommended when running large scale Hadoop deployments. Provided measurements will serve as significant data point in determining wether or not the selected technologies are in fact scalable or not.

Chapter 6 will gather and summarise all recommendations gartered from implementing and tuning the system as if it was an in-production running system. Lastly it will judge wether or not the taken aproach seems to be a feasible one to pursue in future applications.

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 5.

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* [DG04] 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 5, 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 [sca] is a Domain Specific Language implemented using the Scala [Ode13] programming languagg. It has been 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 does not stand on it's own, as it is only a thin layer built on top of Concurrent Inc.'s [con] Cascading library, which 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.

2.3. Apache HBase

HBase is a column-oriented database [Wik] 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 – because it's Tables are layed out as Sequence Files that are a very performant way to store data (such as rows of a big table) on Hadoop's File System. Detailed investigation in Hadoop as well as Sequence File performance will be provided in a later section in Chapter 5.

2.4. Scala

Scala is a functional *and* object-oriented programming language designed by Martin Odersky [Ode13] running on the Java Virtual Machine. It was selected selected as primary implementation language for this project (although 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* (both introduced in this section).

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 [CH73] based concurrency for Scala (and Java) applications. Using this concurrency model can be best explained using two rules "share nothing" and "communicate via messages" – both steming from the programming language Erlang [erl], and indeed both Erlang and Akka provide the same concepts and levels of abstraction. The general gain from using this model is that one is oblivious to *where* an Actor (executor of code / reciver of messages) actually is running – in terms of "in local JVM" or "remotely, and the message will be delivered via TCP/UDP". Thanks to this not-knowing, it is trivial to scale such applications horizontally, since the code does not need to change when moving from 1-node implementations to clustered environments.

For the sake of this thesis, Akka has been used both in local (in-jvm) parallel execution as well as clustered deployment (using Akka's built in clustering module), in order to balance the workload generated by actors across the entire cluster. Details on scaling Akka clusters have been provided in Section 5.2.

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 [Zau10].

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*. During the work on this thesis the source code of phash was slightly modified (in agreement with the software's license) to be adjusted to work better by being called from Java applications.

As the goal of this thesis is not researching such algorithms, but focusing on scalable image analysis computations in distributed systems, it was deeped apropriate to use an existing perceptual hashing solution.

2.7. Chef

OpsCode Chef is a set of tools aimed at automating server configuration and provisioning. Using it enabeled automating spinning up new servers which was a very large part of the work on this thesis, and also one such automation was prepared, it could also be applied to different cloud service provides – which lead to exploring local virtual machines (Vagrant and VirtualBox) as well as Amazon's EC2 public cloud offering until lastly satteling for good on using Google's Compute Engine public cloud offering. The use of Chef enabled not spending tedious hour of reinstalling the cluster each time anew, but allowed to "cook" given virtual machines into the required state (that is, installing Hadoop, Hbase, Java and all the other dependencies needed for running the system implemented during this work).

Appendix A features a detailed overview and guide on the details on how Chef was used in this thesis, and what steps had to be taken in order to prepare the "recipes" it works on to be able to provision Hadoop automatically to any given GNU/Linux running instance.

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.

3. System 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.

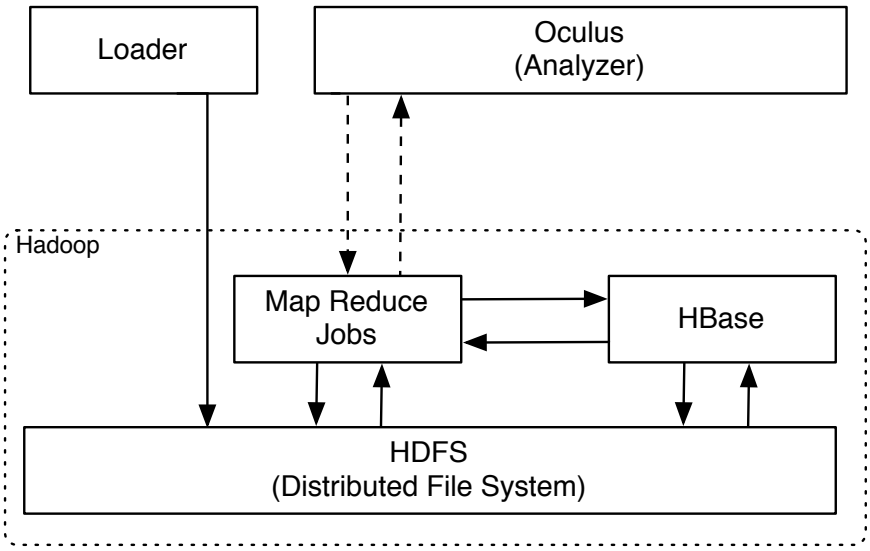


Figure 3.1: High level overview of the system’s architecture

3.1. Loader

The Loader component is responsible for obtaining as much as possible ”reference data”, by which video material from sites such as *youtube.com* or video hosting sites is meant. 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 the Loader system will be referred to from here on 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 [JB13] Actor Model abstractions (and *remoting* module [Inc13]), 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.

The following Actor descriptions are aimed at explain

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 5.1.1.

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.

sends:

0 or 1 – `ConfirmUpload(localFile: java.util.File)` – sent back to the sender that requested the video to be uploaded.

Using these Actors and protocols between them, the application is able to progress safely without the possibility of getting stuck in a dead (or live) lock.

While discussing there protocols one should also mention that the messages that can be received and sent by Actors are not possible to verify using static typing – an Actor can receive a message of "Any" type, and in case of not being able to act upon it - it will drop the message, but the delivery still happens. This is an inherent property of the Actor Model of concurrency – which is why in such systems explaining the flows of mesages and Actors present in the system is of such importance. Having this in mind, the next Section (3.1.2) will focus on explaining one such message flow within the system.

3.1.2. Obtaining reference video material

This section wills 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.

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), a routing strategy called "Smallest Inbox Routing" was used. 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).

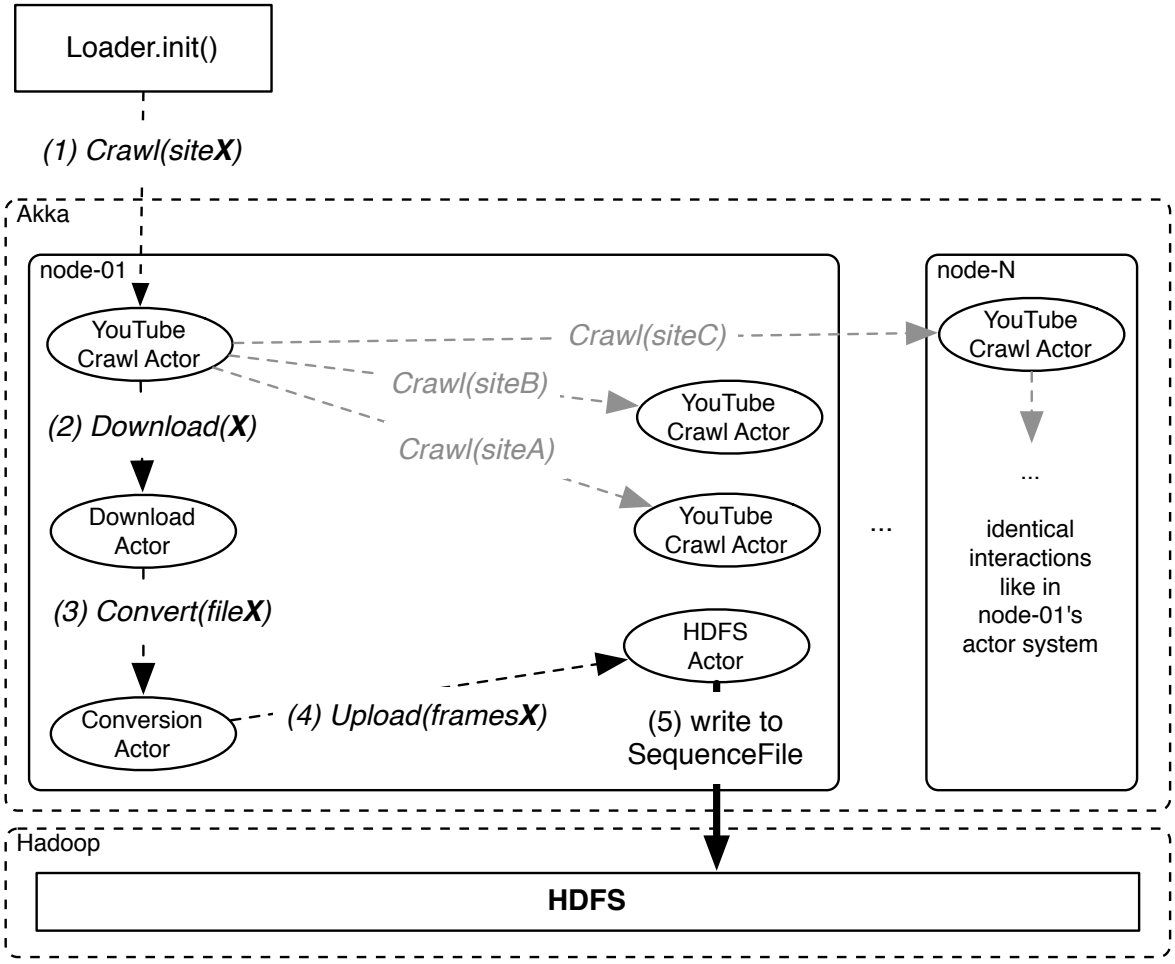


Figure 3.2: Overview of messages passed within the Loader's actor system. Greyed out messages are also sent, but are not on the critical path leading to obtaining material from *siteX* into HDFS.

Step 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` [ffm] 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 5.1.1.

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.

3.1.3. Loader design summary

As was explained in the previous sections the Loader is designed as fully asynchronous system, composed of Actors performing very specific tasks. All communication between the Actors is implemented as message passing, which provides so-called ”location transparency” between the Actors – this property has been utilised to spread the work-load between multiple nodes in a clustered enviroment, enabeling the work to be completed faster than only utilising a single node.

A detailed discussion of the Cluster’s scalability and patterns applied in order to provide ad-hoc joining of nodes to the computation cluster can be found in Section 5: ”Scaling out the Ator system based Cluster”.

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.

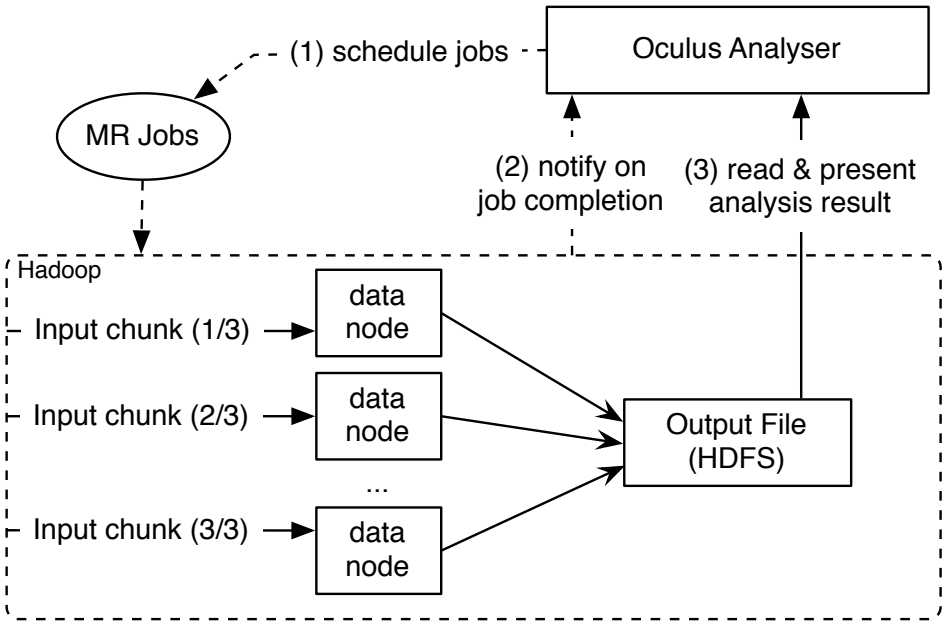


Figure 3.3: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes the *name-node* memory problems.

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. Defining Map Reduce Pipelines using Scalding and Cascading

The language and for implementing these processing pipelines used in this thesis was Scalding – Twitter's home grown Scala DSL, which was already briefly introduced in Section Section 2.2. The choice of using Scalding, and not Hadoop's plain Map Reduce API was mostly dictated by understandability and ease of modifying complicated job pipelines.

Comparison of example Job implemented using Plain Hadoop and Scalding

This section aims to demonstrate what gains using Scalding over Hadoop yields, by showing the smallest possible example of an Map Reduce Job, implemented using both plain Hadoop APIs as well as Scalding. One should remember that Scalding is not a different framework than Hadoop, and in the end it will produce the exact same computations and classes (one instance of an `Mapper` and one instance of an `Reducer`).

The example shown on the below listings is a Hadoop equivalent of what programming languages aim to demonstrate using "Hello World!" applications – it is a minimal piece of code showing the working of the given technology. The task to solve is defined as: "*Given a huge input text file, count the occurances of each discrete word, and return a summary of these*". As the aim of Hadoop is to leverage parallel computing, the classic method of implementing this task is to emit tuples of `(word, 1)` for each word, then relying on Hadoops internal mechanisms to group such tuples together by the first value (the `word`), and reducing these into a tuple of `(word, sum(1,1,1,...))` during the Reduce step.

The difference in complexity should be obvious by comparing the Listing 3.1 which represents the Map and Reduce classes which are used to represent the respectful Map and Reduce functions in the Java API, with Listing **??** which is the complete code for a job performing the exact same in Scalding's Scala Domain Specific Language.

Listing 3.1: Word Count example Job, implemented using plain Java Map Reduce API

```
1  public class Map
2     extends MapReduceBase
3     implements Mapper<LongWritable, Text, Text, IntWritable> {
4
5   private final static IntWritable one = new IntWritable(1);
6    private Text word = new Text();
7
8    public void map(LongWritable key,
9                Text value, OutputCollector<Text, IntWritable> output,
10               Reporter reporter) throws IOException {
11    String line = value.toString();
12    StringTokenizer tokenizer = new StringTokenizer(line);
13    while (tokenizer.hasMoreTokens()) {
14     word.set(tokenizer.nextToken());
15     output.collect(word, one);
16    }
17   }
18 }
19
20 public class Reduce
21    extends MapReduceBase
22    implements Reducer<Text, IntWritable, Text, IntWritable> {
23
24  public void reduce(Text key,
25                Iterator<IntWritable> values,
```

```
26                OutputCollector<Text, IntWritable> output,
27                Reporter reporter) throws IOException {
28   int sum = 0;
29   while (values.hasNext()) sum += values.next().get();
30   output.collect(key, new IntWritable(sum));
31  }
32 }
```

Listing 3.2: Simplest Scalding job used in Oculus – each frame perceptual hashing

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

As seen on the previous listings, Scalding provides much more concise code, and thanks to this enables developers to focus on the algorithm, and not the boilerplace accompanying these kinds of applications.

Defining advanced Job Pipelines using Scalding

Scalding also provides powerful facilities for pipeline building, where by ”Pipeline” we refer to a series of Jobs that use the output of a previous Job as the input for the next one. It is worth mentioning that this is not something plain Hadoop APIs are able to provide easily, and one would have to implement logic in Jobs that would store the intermediate output in a directory and await the Job’s completion, to then manually start the second job (by running it’s main method).

Building Pipelines has two major styles in Scalding: explicit ”next Job” definitions, as well as implicit dependencies on results of Jobs. We will investigate both styles in this section – starting with the explicit style, as it is more similar to the manual style of doing things.

Listing 3.3: Explicit ”next job” definition within a Scalding Job class

```
1  case class FirstJob(args: Args) extends Job(args) {
2    val in = args("in")
3    val out = in + ".out"
4
5    Tsv(in).read.write(Tsv(out))
6
7    def next = Some(SecondJob(Args("--in", out)))
8  }
9
10 case class SecondJob(args: Args) extends Job(Args(out)) { /*...*/ }
```

Listing 3.3 shows how one can use Scalding to combine two Jobs using Scalding’s DSL. The `SecondJob` depends on the output of the `FirstJob` (which in this case only copies the input to another file on HDFS), since the `next` method is defined within the first Job, we have it’s parameters available and can construct the next Job in the pipeline, providing it with it’s required `in` parameter.

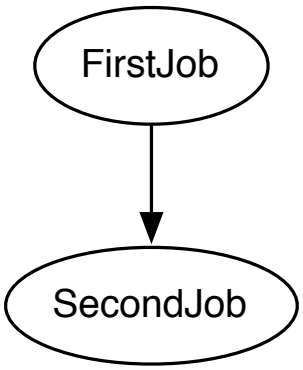


Figure 3.4: Simplest Job Pipeline, with SecondJob depending on the output of FirstJob (each circle being one Map Reduce Job).

Such pipeline can be visualised as seen on Figure 3.4, which is an output generated by using the graphviz tool (as in "graph visualization"), applied to a file describing the graph as described in the DOT format. The DOT file corresponding to the graph depicted on Figure **??** is shown in Listing 3.4. DOT descriptions of pipelines are in common use among profesionals designing such systems, and can also be generated automatically by Scalding – which is tremendously useful when working with very long pipelines.

Listing 3.4: Textual description of graph on Figure 3.4, using the DOT graph description language.

```
digraph G {
  1 [label = "FirstJob"];
  2 [label = "SecondJob"];

  1->2
}
```

The second way of building up pipelines is inside one Scalding Job file. Because Scalding's rich set of operations, what may look as simple on the surface (in the pipeline definition) may have to boil down to multiple Map Reduce Job executions. One obvious case that will cause one Scalding Job to produce multiple Map Reduce Jobs is using data from two separate sources and joining them together. It should be noticed that the notion of "join" (as known from relational databases) is something both very powerful and very foreign to Hadoop – as it is only designed to deal with data in a batch-processing fashion. Scalding allows to express join operations trivially in the the definition file, but it's execution actually is quite complicated and forces all the data from one side of the join to be loaded into Mappers participating in the Job's execution.

Listing 3.5: Scalding job, reading data from 2 sources and joining them on dominantColor, producing 3 Map Reduce Jobs

```
class CompareByDominantColor(args: Args) extends Job(args) {
  // newly analysed video
  val analysedMovieFrames =
```

```
4     WritableSequenceFile(input, ("key", "value"))
5     .read
6     .rename("key", "frame")
7     .map(("frame", "value") -> ("dominating", "red", "green", "blue")) {
8        p: (Int, BytesWritable) => calculateHistograms(p._2)
9     }
10
11   // reference database
12   val referenceFrames =
13     HistogramsTable.read
14     .rename("dominating", "refDominating")
15
16   // join
17   referenceFrames.joinWithSmaller(analysedMovieFrames,
18                                   "dominating" -> "refDominating")
19     // operations on joined dataset
20 }
```

Listing 3.5 showcases a simple join operation, which is the core of the algorithms implemented in this thesis. This pipeline definition compiles down into 3 Map Reduce Jobs, because it reads from 2 separate data sources: from a sequence file that we just started processing (this is the movie this job aims to compare to the referece data set), and from the `HistogramsTable` which is the reference dataset containing refernece data calculated previously for all already processed movies in the reference database. The `HistogramsTable` refers to an *HBase Table*, and during the Job's executionw will trigger (in this case) a full-scan over the entire "histograms" table stored in HBase - in practice it was feasible to sample down the sample count obtained from HBase, by applying the `.sample(75.0)` operation to the larger data set. Figure **??** shows the dependencies as modeled by this definition.

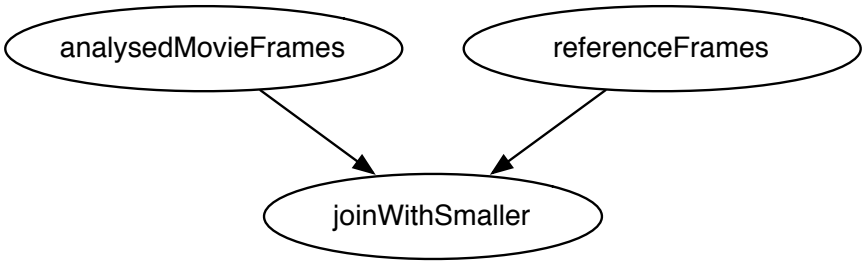


Figure 3.5: Join operation, forcing the Pipeline to consist of 3 Map Reduce Jobs (depicted as circles)

The most important thing to notice is that the dependencies, and temporary output directories which are used to pass the data around between the first two Jobs to the 3rd "Join Job" were not explicitly modeled, but infered from the use of the `job1.joinWithSmaller(job2, "key1" -> "key2"` operation. Cascading will also take case of deleting these temporary data directories after the job has successuly completed – another great gain when compared to manual Map Reduce Pipeline implementations.

This section has highlighted how Map Reduce Pipelines can been implemented using Scalding, and

have introduced the basic concepts required in order to implement algorithms that can be seen in the following sections.

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 [**?**].

Figure **??** 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 3.6 Figure **??** would look like Figure **??**.

Listing 3.6: Example of Scalding Pipeline exported from the Cascading FlowPlanner using the DOT format

```
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']"];
# ...
}
```

The graph represented on Figure **??** 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.

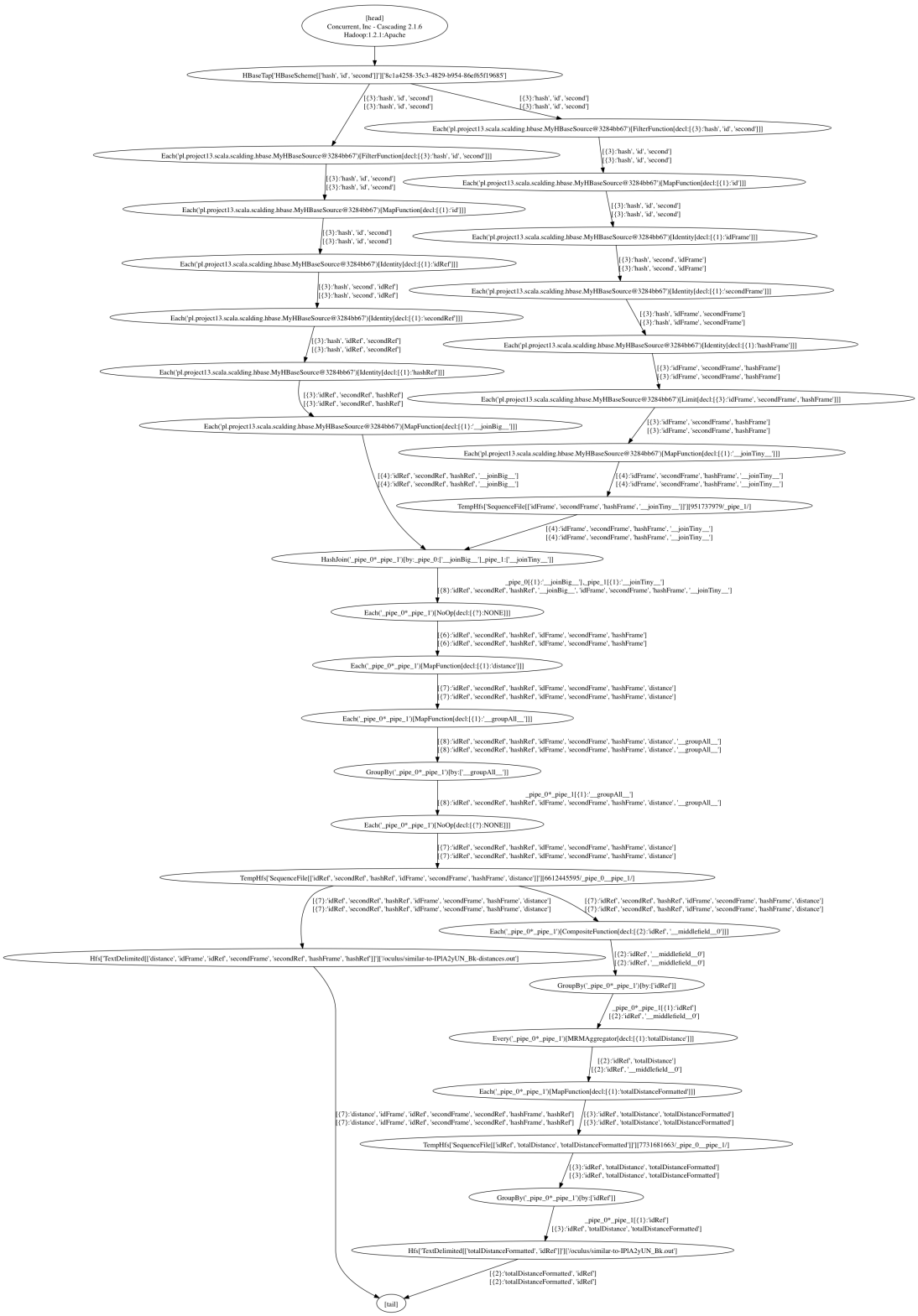


Figure 3.6: This flow represents the most longest Pipeline preset in Oculus – finding which movies are similar to the current one, ordered by ranking. It is constructed from 5 Map Reduce jobs.

ą

use I, not WE this is not a blog post...

needs

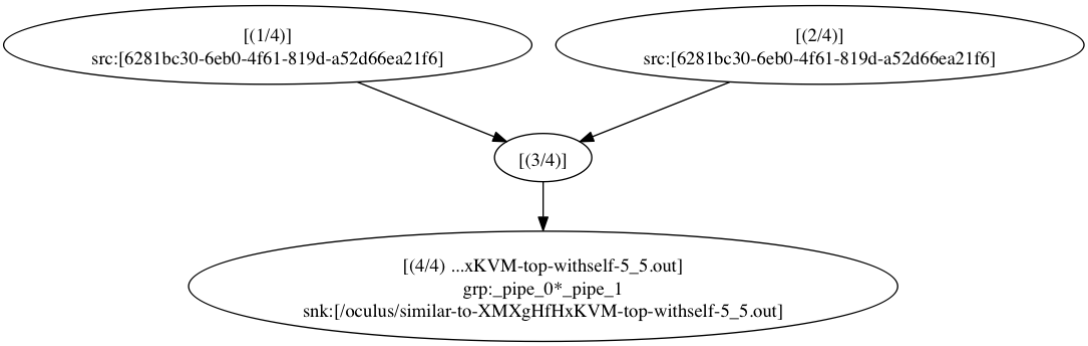


Figure 3.7: A graph representation of the Map Reduce jobs that have to be run in order to complete the pipeline.

4. Practical examples of distributed media analysis

This chapter aims to provide tangible examples of how the previously described system preformed and what kind of information was extracted from the reference database when trying to match against disorted media inputs.

The following two sections will focus on practical examples of potential ”attacks” (disortions) applied to the input media data, and how the proposed system has handled it. The examples have been selected to highlight the two major problems the system has to handle – distorted image data and time shifted data.

In Section 4.1 video material will be submitted to the Oculus Analyser in order to find it's corresponding ”mirrored” counter-part. This example will also be used to highlight the tremendous possibilities that lie within data *pre-processing* that are applied within the proposed system, and if needed could be expanded even more in order to quicken the initial response time.

In Section 4.2 an extracted scene will be positioned within an existing video in the reference database. This problem turns out to be non-trivial because of different frame-rates of supplied material, thus rendering methods similar in concept to sub-string search could not have been applied efficiently. The section explains the algorithms applied instead, and showcases an example case.

The frames used to illustrate the experiments stem from the movie ”Big Buck Bunny” [Fou08] which has been created by the Blender Foundation [ble] and released under the Creative Commons license [Com01].

4.1. Mirrored video detection

In order to test the system in scenarios of image disortion the example case of ”mirrored” video material was used. This case is fairly popular among material uploaded to YouTube, so outside of the ease of preparing test data, it is also a valid real-life scenario of one might encounter.

4.1.1. Detecting other kinds of distorions in video material

Of course, other kinds of disortions could be applied to a video, such as comparing a high-quality video with a low-quality counterpart or simple coloring filters. While being outside of the scope of this thesis, the basic concepts and framework proposed would easily be able to incorporate more scoring functions into the Map Reduce pipeline that would help determine matches between even such disorted video materials.

(a) Original frame (b) Mirrored frame

Figure 4.1: Example of original and mirrored frame

An interesting example disortion found commonly in many materials is slight changes in the color hue or white balance of the histogram

4.2. Scene detection

This scenario can be explained as trying to find out *where* (if at all) a scene takes place in a movie known to the reference database.

Although on the sufrace the general problem statement is not so different than substring search, which is a known and well researched topic in computer science. In sub-string search algorithms like the Knuth–Morris–Pratt [kmp] or the Boyer-Moore [RSB77] algorithms leverage that the ”matching” either will apply, or will not in order to increase search speed in the worst case to still linear time. However, these methods can *not* be directly applied to the problem specified in this section – because of the dissortions in source and reference video material as well as the possibility of cross-matches when a movie is built up from multiple short scenes from other movies – a typical example here would be ”flash-back” scenes or ”top 10” movies where before the last top-3, the movie would quickly go over already shown frames of scenes. Another problem adding to the dissortions is frame rates of reference data vs. an analysed video fragment – even a slight missmatch (30FPS vs. 25FPS) would render the substring search algorithms not usable for this concrete example.

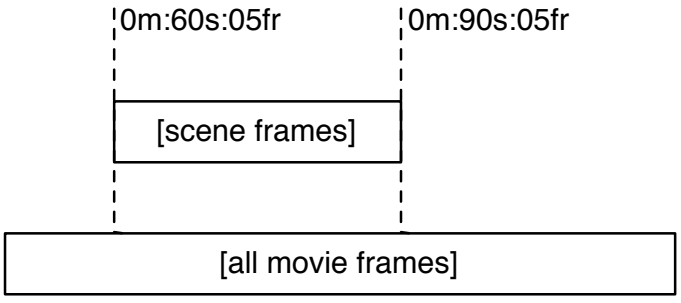
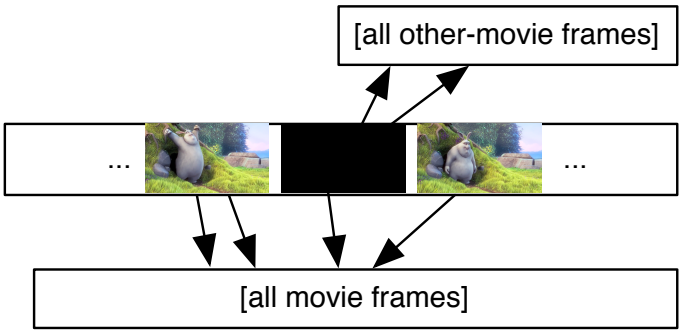


Figure 4.2: Visual representation of the goal of this example application.

Instead, a more statistical aproach was taken. In which a frame is compared with it’s set of ”potential match candidates” which are determined by very coarse filtering of the reference data set by bucketing

certain criteria of their histograms such as ”dominating red” or ”dominating blue” (this classification is prepared on the reference data beforehand – during importing into the system).

mention somewhere before that we really extract data like ”mostly red”



finish this scenario

Figure 4.3: One frame may potentially match multiple reference frames. The final most probable matching scene is determined by aggregating data the direct frame-to-frame matches.

5. Cluster scaling and performance analysis

This section will focus on analysing as well as presenting improvements to the cluster deployment which can and have been applied to the Hadoop cluster leveraged by the Analyser application presented in previous chapters.

In Section 5.1 several challanges and typical problems with scaling Hadoop clusters will be presented, and then followed up by solutions applied during the work on this thesis.

Section 5.2 will briefly explain scalability concerns related to an Akka based cluster deployment, yet as this component has not been as critical to overal system performance as the Analyser and Hadoop cluster, the Akka cluster has been deemed ”good enough” for the scope of this paper.

Lastly Section 5.3 will summarise the findings from the scaling experiments conducted in the previous sections, by stating general recommendations and hints for scaling systems that process larget amounts of data.

5.1. Scaling the Analyser’s Hadoop Cluster

This section will focus on analysing and tuning the various settings of the Hadoop cluster deployed for the previously described Analyser application. Subsections focus on tuning the cluster on a setting–by–setting basis yet the tuning will always be enforced by a business need, which in this case will be represented as the need to speed up processing of the map reduce pipelines producing results which were explained in Chapter 4.

5.1.1. Storing images on HDFS, while avoiding the ”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 unadressed. This section will focus on explaining the problem and what steps have been taken to prevent it from manifesting in the presence of millions of ”frame-by-frame” pieces of data.

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.

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 worrying, 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 5.1 shows the structure of a block storing only one frame of a movie.

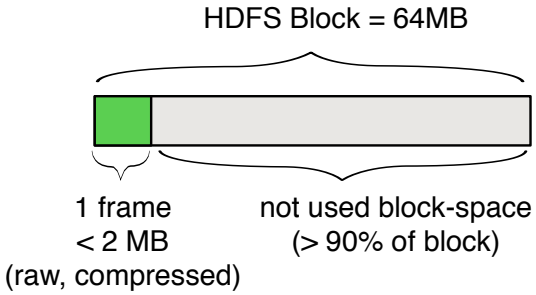


Figure 5.1: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes *name-node* to store 1 block entry in memory.

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 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.

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 [Had12] but they all are *binary header-prefixed key-value formats*, as visualised Figure 5.2.

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.

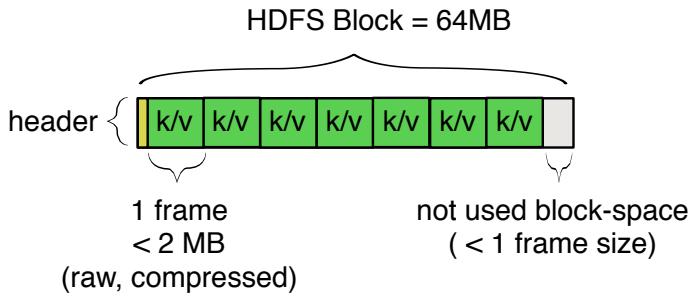


Figure 5.2: A SequenceFile allows storing of multiple small chunks of data in one HDFS Block.

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.

5.1.2. Tuning replication factors

One of the many tuneable aspects of Hadoop deployments that can have a very high impact on the clusters performance is the *replication factor*, which stands for "the number of datanodes a piece of data is replicated to". This section will explain in detail how a tuning this factor, and leveraging Hadoop's scheduling mechanisms can be tweaked to trade of storage space (higher replication factors) to faster execution times of jobs.

Hadoop's primary strength in big data applications lies within leveraging *data locallity* whenever possible. The concept of data locallity means that instead of moving the data around in the cluster, to a node where the application is running, the Task Scheduler will try find such "map slots" (multiple such slots can be assigned to one data node) that the data the job needs to process will be local to the node the slot resides on. Effectively this means that application code (jar and class files) will be sent to the executing server, and not the inverse. The rationale behind this measure is that the amounts of data are way bigger than the size of applications in these kinds of systems – thus, avoiding to move the data can save both costs and precious time.

While data locallity is a *priority* for the scheduler, it is by no means a hard requirement. In a scenario outlined in Figure **??** a job has been submitted to a 3-node cluster. The replication factor in the cluster is set to 2 – which can be noticed by the number of times each piece of data is replicated among the datanodes. In the absence of any other jobs scheduled on the cluster, the fair–scheduler will decide to use the 3rd data node in order to accelerate the processing of the job, even though it does not have the required piece of data located on it (splits of A). Replicating the data over to datanode-3 is quite costly, and even though it may speed-up the total compute time, we loose time on transfering the data in an ad-hoc fashion.

By tweaking the replication factor for the given file, which we expect to be needed on more nodes, we can speed up the total compute time of a given job. In order to change the replication factor of a given

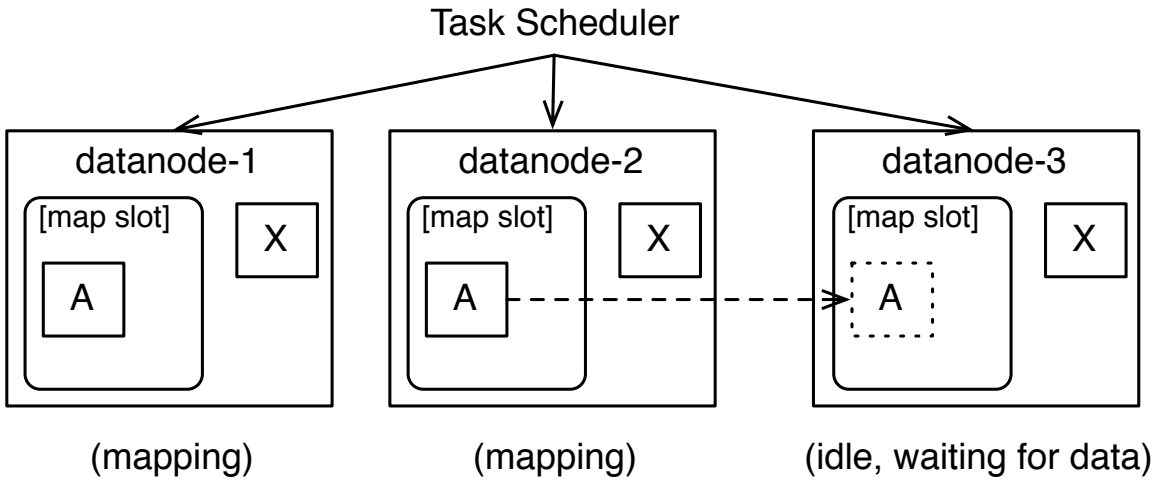


Figure 5.3: Three node cluster, with idle 3rd node; Scheduler will replicate data A while running the Job, in order to start a task requireing A on the 3rd idle node.

path, one can use either the Java APIs or the hadoop command line tool, as shown in Listing 5.1.

Listing 5.1: Explicitly changing the replication factor on a path using command line tools

```
1 hadoop dfs -setrep -R -w 3 /oculus/source/e98uKex3hSw.mp4.seq
```

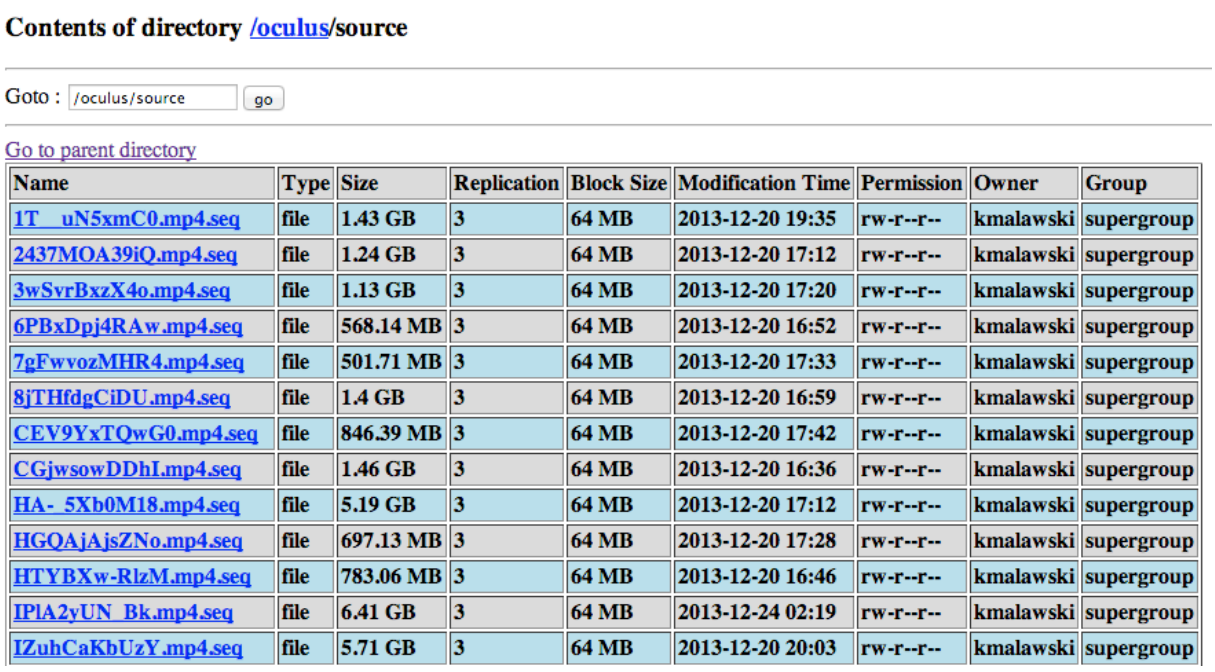
By increasing the replication factor of paths that we know they will be used in many mappers, we can increase the number of data-local slots available to the scheduler, and avoid having to migrate the data in an ad-hoc fashion. Of course, the tradeoff is requireing even more disk space in the cluster, but it is well worth considering to raise the replication factor of a path while it is "hot", and lowering it afterwards.

The replication factor of a file (or path) is such an important value, it's usually always displayed along side any file listing within the Hadoop UIs (see Figure **??**) as well as command line tools. It should also be noted that it is impossible to set the replication factor to a higher number than there are datanodes present in the cluster – as the replication requirement would not be possible to be fulfilled.

5.1.3. Tuning the Cluster's size, in conjunction with replication factors

Hadoop aims to deliver on the promise of "nearly linear horizontal scalability", which means that speed-up expirienced from adding more nodes to the cluster should impact the processing times positively in a linear fashion. Of course, adding more nodes also means that operational costs are higher, so one has to ballance the number of nodes with their fine-tuning as well as project needs. In order to test the systems behavior, the cluster was configured with 1 map slot on each datanode, and was initially running using 3 datanodes. This section aims to verify the horizontal scalability of the produced cluster, in the case of CPU intensive tasks – such as computing phashes and histograms from movies (the pre-processing step as explained in Chapter 4.

Because the Map Reduce framework relies on parallelising computing of the map function supplied by the user, the goal of the cluster administrator should be to enable the maximum number of map slots.



Contents of directory /oculus/source

Goto : /oculus/source go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
1T__uN5xmC0.mp4.seq	file	1.43 GB	3	64 MB	2013-12-20 19:35	rw-r--r--	kmalawski	supergroup
2437MOA39iQ.mp4.seq	file	1.24 GB	3	64 MB	2013-12-20 17:12	rw-r--r--	kmalawski	supergroup
3wSvrBxzX4o.mp4.seq	file	1.13 GB	3	64 MB	2013-12-20 17:20	rw-r--r--	kmalawski	supergroup
6PBxDpj4RAw.mp4.seq	file	568.14 MB	3	64 MB	2013-12-20 16:52	rw-r--r--	kmalawski	supergroup
7gFwvozMHR4.mp4.seq	file	501.71 MB	3	64 MB	2013-12-20 17:33	rw-r--r--	kmalawski	supergroup
8jTHfdgCiDU.mp4.seq	file	1.4 GB	3	64 MB	2013-12-20 16:59	rw-r--r--	kmalawski	supergroup
CEV9YxTQwG0.mp4.seq	file	846.39 MB	3	64 MB	2013-12-20 17:42	rw-r--r--	kmalawski	supergroup
CGjwsowDDhI.mp4.seq	file	1.46 GB	3	64 MB	2013-12-20 16:36	rw-r--r--	kmalawski	supergroup
HA-_5Xb0M18.mp4.seq	file	5.19 GB	3	64 MB	2013-12-20 17:12	rw-r--r--	kmalawski	supergroup
HGQAjAjsZNo.mp4.seq	file	697.13 MB	3	64 MB	2013-12-20 17:28	rw-r--r--	kmalawski	supergroup
HTYBXw-RlzM.mp4.seq	file	783.06 MB	3	64 MB	2013-12-20 16:46	rw-r--r--	kmalawski	supergroup
IPlA2yUN_Bk.mp4.seq	file	6.41 GB	3	64 MB	2013-12-24 02:19	rw-r--r--	kmalawski	supergroup
IZuhCaKbUzY.mp4.seq	file	5.71 GB	3	64 MB	2013-12-20 20:03	rw-r--r--	kmalawski	supergroup

Figure 5.4: HDFS on-line browser, running on datanode (port: 50075), displaying replication factors of files (4th column)

A map slot is defined as one "slot" in which the task scheduler may allocate work, in order to process a part of the map computation. The same can be specified for the reduce step, in which case one refers to *reduce slots*.

Growing the cluster

With the base size of the cluster being 3 nodes (with one virtual machine hosting the namenode as well as a datanode, and the rest hosting only datanodes), this section aims to determine the horizontal scalability of the cluster, by sheer adding of datanodes to the hadoop cluster.

Adding nodes to the cluster is a very simple operation, and can be performed without any disruption of the already running cluster. During the tests performed for this section, additional VMs have been provisioned and added to the cluster by using the commands shown in Listing **??**. It should be noted that the provisioned VMs would re-use an existing snapshot image of an instance prepared using OpsCode Chef, which is an configuration provisioning tool explained in detail in Appendix A. The usual time from starting a new node on Google Compute Engine, and it starting to participate in the work distribution of the Hadoop cluster was *less than 1 minute* (including the time to provision the fresh virtual machine)!

,

Listing 5.2: Complete listing of adding a new worker node to the cluster, using GCE

```
1 // provision new instance
2 gcutil --service_version="v1" --project="oculus-hadoop" adddisk
    "oculus-4b" --zone="us-central1-a"
    --source_snapshot="oculus-slave-snapshot"
3
```

```
4  gcutil --service_version="v1" --project="oculus-hadoop" addinstance
       "oculus-4b" --zone="us-central1-a" --machine_type="n1-standard-1"
       --network="default" --external_ip_address="ephemeral"
       --service_account_scopes="https://www.googleapis.com/auth/..."
       --tags="hadoop,datanode,hbase"
       --disk="oculus-4b,deviceName=oculus-4b,mode=READ_WRITE,boot"
       --auto_delete_boot_disk="true"
5
6  // obtain ip address
7  gcutil getinstance oculus-3b | grep ip
8  // ip          10.240.80.181
9  // external-ip 146.148.47.191
10
11 // add internal ip to namenode masters config
12 gcutil ssh oculus-master
13 echo "10.240.80.181" >> /opt/hadoop.1.2.1/conf/slaves
14
15 // start workers on added nodes
16 /opt/hadoop-1.2.2/start-all.sh
17
18 // add node aliases to local hosts
19 echo "146.148.47.191 oculus-3b" >> /etc/hosts
```

The job used to measure the impact of adding new nodes was the most CPU intensive task present in the Oculus workflow, which is: computing the perceptual hash of each frame of a given movie. The movie selected for the process was the previously introduced "Big Buck Bunny" movie, amounting a total of 6.38 GB of frame data to process, split among 103 HDFS Blocks (each roughly 64 MB in size). Thanks to this large number of blocks, the task scheduler should be able to efficiently distribute the work to even large numbers of worker nodes. In fact, as visible on Figure **??** with 10 mappers (10 datanodes, with 1 map slot each) are executing tasks from the same job in parallel, causing an obvious speedup in Map comptation.

Before explaining the results of the performed scalability tests, one more term needs to be introduced. Tasks can be executed in a number of different ways – that is, they may be executed strictly local to the data they work on, or just "near the data the task requires" or really "far away from the data the task requires" – these intuitive definitions have their named counterparts which are measured and reported for every run of a Map Reduce job, namely those types of Tasks are:

- **Data Local Task** – the Task is executed on the same datanode on which the data it requires resides. No data transfer between nodes is required for the task to start. This is the optimal type of Task, and one should aim at maximising their number during an execution of Map Reduce jobs.

- **Rack Local Task** – the Task is executed on a datanode that is located on the same rack (in the datacenter) as the node that the data it requires resides. This kind of task will require the data to be transfered between the two hosts, but since they are located on the same rack the transfer cost is still relatively small.

Hadoop map task list for job_201312310015_0049 on oculus-master

Running Tasks

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_201312310015_0049_m_000004	98.93%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:268435456+67108864	22-Apr-2014 01:21:35			19
task_201312310015_0049_m_000010	79.89%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:671088640+67108864	22-Apr-2014 01:22:23			0
task_201312310015_0049_m_000011	20.97%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:738197504+67108864	22-Apr-2014 01:22:44			0
task_201312310015_0049_m_000012	25.10%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:805306368+67108864	22-Apr-2014 01:22:45			0
task_201312310015_0049_m_000013	16.31%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:872415232+67108864	22-Apr-2014 01:22:46			0
task_201312310015_0049_m_000014	12.73%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:939524096+67108864	22-Apr-2014 01:22:46			0
task_201312310015_0049_m_000015	7.78%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:1006632960+67108864	22-Apr-2014 01:22:46			0
task_201312310015_0049_m_000016	15.05%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:1073741824+67108864	22-Apr-2014 01:22:47			0
task_201312310015_0049_m_000017	3.38%	hdfs://108.59.81.83:9000/oculus/source/YE7VzlLtp-4.mp4.seq:1140850688+67108864	22-Apr-2014 01:22:49			0
task_201312310015_0049_m_000018	0.00%	initializing	22-Apr-2014 01:22:52			0

Figure 5.5: Fragment of Web-UI interface displaying the progress of map tasks in the cluster consisting of 10 physical nodes, with one map slot each.

– **Remaining Tasks** – tasks that are not Data Local do not have a name of their own, and are not reported directly. Instead one aims to maximise the number of Data and Rack Local tasks, with the rest being simply `notLocalTasks = totalTasks - dataLocalTasks - rackLocalTasks`. These tasks require transfering the data across the network for the job to start, and should be avoided at all costs – in our use-cases it was possible to avoid triggering even one of these tasks.

The first row in Table **??** represents the theoretical time to execute the job using only one node – the aproximated time to complete the job using one thread (mapper) was calculated by calculating the average time of computing one split of data, which is equal to: *47 seconds*. Other time characteristics of computing one task are listed, for reference, in Table **??**.

Characteristic	Value (seconds)
min	41
max	71
avg	47.41
stddev	4.33

Table 5.1: Time characteristics of executing one Task of the analysed job.

The experiment was then expanded to more nodes, and afterwards the replication factor was increased to increase the chance of triggering Data Local tasks. Results of the experiment are listed in Table **??**.

Analysing Table **??** yields very interesting results. The most interesting metric in our case if of course the absolute speed-up of the process, yet the relative speed-up (as measured between previous and current configuration) is also quite interesting – especialy in cases when the replication factor was modified.

Cluster configuration		Tasks executed		Performance		
Nr of nodes	Replication	Data local	Rack local	Time	Speed-up	Abs. speed-up
1 node (simulated)	1	103	0	aprox. 87 mins	–	–
3 nodes	3	103	0	27 mins, 14 sec	3.19	3.19
4 nodes	3	89	14	21 mins, 53 sec	1.25	3.98
7 nodes	4	67	36	12 mins, 37 sec	1.73	6.90
10 nodes	4	54	49	9 mins, 1 sec	1.40	9.65
	6	99	4	8 mins, 49 sec	1.02 (1.43)	9.87

Table 5.2: Computation speed-up by adding nodes, and increasing replication factors, leading to increased data locality for exacuted jobs.

The first notable and interesting difference is of course between running the job on 1 node to 3 nodes, with the replication factor equal to the number of nodes to the cluster. This has the obvious effect of all tasks being executed local to the data they require (as each node has all the data). The total speed-up is around 3, which would indicate linear scalability in this case. Yet becuase of still small numerf of nodes, and everything being executed locally, this case is not very interesting from the ”introduction of cluster communication overhear” perspective.

The next notable effect demonstrated during this experiment is visible when scaling the cluster to 4 nodes, without increasing the replication factor (which stayed at 3). This forced the cluster to execute Rack Local tasks for the first time. The scheduler was forced into scheduling 14 tasks on nodes that did not have the data locally available, although it managed to schedule them on Rack Local tasks – which still is an acceptable choice for most computations.

After scaling the cluster to 10 nodes, and updating the replication factor to 4 – specifically chosen because it being ”slightly bellow half the number of the nodes”, one can observe a significant drop in tasks being executed Data Locally – this is because the scheduler has more nodes available, and since they are idle, it decides to use them instead of keeping them idle – it will do so, even if there are no Data Local slots available. In the end it still results in an `1.4` speed-up as related to the 7 nodes scenario, even if the number of Rack Local taks has incresed by a factor of `3.5`.

The last, and perhaps most interestning measured change to the cluster involved increasing the replication factor on an 10 nodes cluster to `6` – so that more than half of the servers would host the same piece of data. After waiting for the data to be propagated (notice that datanodes undergoing heavy migration can appear unavailable to the cluster), the job was started again. This time, even though the relative number of nodes that have the data locally was not so much different (”slightly more thank half of the nodes”) as in the 7 nodes scenario, it's highly interesting to see that `99` out of `103` (96%) tasks were executed as Data Local tasks. While the change in the numbers of Data Local tasks executed seems fantastic, the difference in relative speed-up between replication factors 4 and 6 in this case in relation to the 7 nodes case was a mere `0.03` precentage points, and around 11 seconds, a speed-up most likely not not worth the added data duplication introduced to the cluster – which also has a financial backlash (more data duplication, resulting in the need of even more disks, resulting in the need of even more servers).

In order to summarise this section, one last observation should be made about Table **??**, namely: in

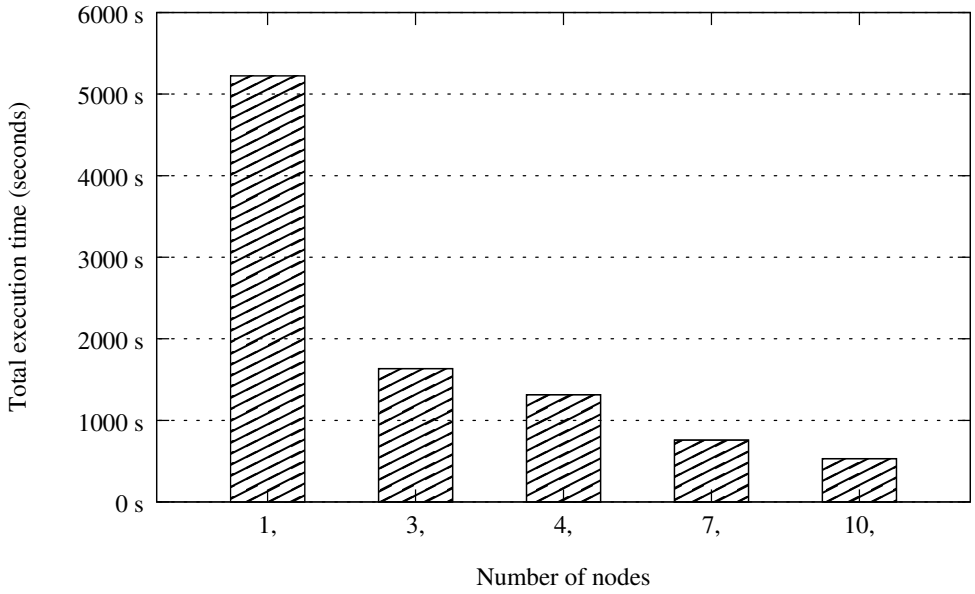


Figure 5.6: Fragment of Web-UI interface displaying the progress of map tasks in the cluster consisting of 10 physical nodes, with one map slot each.

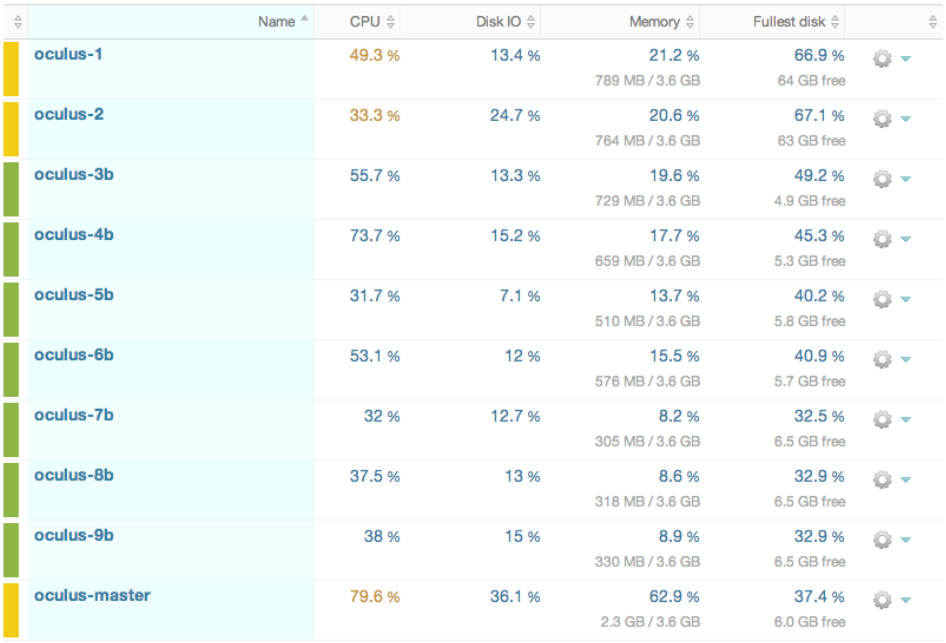
respect of adding more nodes to the Hadoop cluster, it seems that it scales nearly linear, as can be seen by comparing the number of nodes compared to the absolute speed-up, for example – when running a cluster with `10` nodes, the absolute speedup in relation to running on one node was `9.87 times`, which is very near to a full 10. Thanks to this experiment one can conclude that (at least to such numbers of nodes) Hadoop is in fact *nearly linearili scalable*, which is considered quite an achievement and would serve the growing-over-time needs of a data processing platform very well.

5.1.4. Tuning cluster utilization through setting map / reduce slot numbers

The last investigated option available for increasing cluster utilisation investigated was the `mapred.map.tasks` setting and it's corresponding `mapred.reduce.tasks`, as hinted by Eric Sammer in [had].

These settings influence the number of Map and Reduce "slots" available for the scheduler on each node. By default these are set to 1 map task for each physical CPU available on the node (as can be seen on Figure **??**, where *oculus-3-cpu* has 2 physical CPUs, and was automatically assigned 2 map and reduce slots). When running multiple example jobs on the cluster, it was discovered that some "low utilisation" jobs would needlessly occupy precious map slots on the cluster – leading to such worst-case scenarios as depicted on Figure **??**, where all nodes (all of which are of type: *n1-standard-1 (1 vCPU, 3.8 GB memory)*) are processing one task and which occupies 100% of their task slots (because each has 1 CPU), yet the task is *not* CPU intensive, which leads to wasting precious CPU time. It would be much more efficient to allow these nodes to run at least 2 map and reduce tasks at the same time - since they they won't be competing for each other's CPU time.

In order to avoid cluster under utilisation as seen on Figure **??** (where 10 nodes are working, yet the vast majority of them only utilises around 30% of their CPU), the number of concurrent tasks to be executed on one node was increased to 2 map and 2 reduce tasks. Literature recommends using `rount(1.5 * nrOfCpus)` tasks per node, yet this setting should be always accustomed to the work-



Name	CPU	Disk IO	Memory	Fullest disk
oculus-1	49.3 %	13.4 %	21.2 % 789 MB / 3.6 GB	66.9 % 64 GB free
oculus-2	33.3 %	24.7 %	20.6 % 764 MB / 3.6 GB	67.1 % 63 GB free
oculus-3b	55.7 %	13.3 %	19.6 % 729 MB / 3.6 GB	49.2 % 4.9 GB free
oculus-4b	73.7 %	15.2 %	17.7 % 659 MB / 3.6 GB	45.3 % 5.3 GB free
oculus-5b	31.7 %	7.1 %	13.7 % 510 MB / 3.6 GB	40.2 % 5.8 GB free
oculus-6b	53.1 %	12 %	15.5 % 576 MB / 3.6 GB	40.9 % 5.7 GB free
oculus-7b	32 %	12.7 %	8.2 % 305 MB / 3.6 GB	32.5 % 6.5 GB free
oculus-8b	37.5 %	13 %	8.6 % 318 MB / 3.6 GB	32.9 % 6.5 GB free
oculus-9b	38 %	15 %	8.9 % 330 MB / 3.6 GB	32.9 % 6.5 GB free
oculus-master	79.6 %	36.1 %	62.9 % 2.3 GB / 3.6 GB	37.4 % 6.0 GB free

Figure 5.7: An example of an underutilised cluster, where each node has 1 map and 1 reduce slot, yet the computed task is not draining the available CPU time, leading to wasting precious compute time.

Name	Host	# running tasks	Max Map Tasks	Max Reduce Tasks
tracker_oculus-1.c.oculus-hadoop.internal:localhost/127.0.0.1:41581	oculus-1.c.oculus-hadoop.internal	0	1	1
tracker_oculus-3-cpu.c.oculus-hadoop.internal:localhost/127.0.0.1:42670	oculus-3-cpu.c.oculus-hadoop.internal	0	2	2
tracker_oculus-master.c.oculus-hadoop.internal:localhost/127.0.0.1:38701	oculus-master.c.oculus-hadoop.internal	0	1	1
tracker_oculus-2.c.oculus-hadoop.internal:localhost/127.0.0.1:35678	oculus-2.c.oculus-hadoop.internal	0	1	1

Figure 5.8: Fragment of Web-UI displaying the connected datanodes. The oculus-3 node is an high-cpu instance, and has more slots than the remaining nodes.

load a cluster is expiriencing. Having this in mind, a scaled down cluster may actually be more efficient on processing such jobs. The changed configuration included 4 nodes, all of which were set-up fo host 2 map slots and 2 reduce slots. The CPU load on the cluster in such configuration, performing the same job as seen on Figure **??**, can be seen on Figure **??**.

It is worth keeping in mind that cost efficiency usually is also an important business need and simply adding more servers sometimes isn't an available option. Even more so, sometimes we may end up over provisioning servers – as seen in Figure **??**, so while simply adding more servers is very tempting and usually works very well on Hadoop clusters (as seen in Section 5.1.3), one should always first try to fine tune the cluster to the specific work-load it is experiencing. In the case shown in this section, a fine-tuned

Name	CPU	Disk IO	Memory
oculus-1	96.8 %	25.1 %	24.6 % 915 MB / 3.6 GB
oculus-2	89.5 %	27.9 %	23.3 % 864 MB / 3.6 GB
oculus-3-cpu	93.2 %	19.3 %	70.7 % 1280 MB / 1810 MB
oculus-master	97.7 %	23.5 %	40.9 % 1520 MB / 3.6 GB

Figure 5.9: Screenshot of New Relic (cluster monitoring software) displaying CPU and memory utilisation during the execution of an CPU intensive map reduce job.

4 node cluster was able to perform almost as good (for this set of jobs), as the under utilised 10 node cluster. In real life clusters will always execute very different workloads at the same time, so it may be very hard to fine tune it for very percise scenarios, yet the effort of changing configurations of cluster members can sometimes prevent the need of adding more servers, so it should be always considered first – unless a long term cluster expansion (e.g. "adding 100 nodes, in the next 2 months") is planned directly.

5.2. Scaling out the Loader's Akka Cluster

Scaling the Loader sub-system in this project was not very challanging because of the Actor System provided by Akka being so efficient where as the tasks performed by a node once it got a "download movie" message being measured in multiple minutes (up to 10 minutes for long movies – since most creative commons licensed movies are not very long).

5.2.1. Scaling out by adding more nodes

The one optimization performed during this work was distributing "download" tasks to different nodes, so that two nodes in the cluster would download movies and upload the raw data into HDFS, for later processing by the Hadoop Map Reduce Jobs. By being mostly responsible for downloading, and uploading large files from YouTube and into HDFS, and only the minority of time being spent on crawling YouTube for additional movies to download scaling out the Loader was not a priority in boosting the overall systems performance.

Nevertheless, using Akka's clustering module it was possible to easily scale out the cluster and add new nodes to it, *without the need of stopping the cluster*. The strategy was to deploy one downloader on each of the nodes, and make the node join the Akka cluster. Other nodes in the system would be notified that a new node has joined and can ask it to perform work. A full example of this workflow is presented in Listing 5.3, where the `YoutubeCrawlActor` is configured to listen for cluster membership changes (on line 6), and then can react on members joining and leaving (lines 14 and 15) by adding and removing them from the RoundRobinRouter (which was explained in detail in Section 3.1. Then, upon parsing

a website and extracting links from it, the Router can be used to evenly distribute work among the registered downlaoder actors (lines 11 and 12).

Listing 5.3: Listening for Cluster events in Akka allows the application to dynamically respond to nodes being added to the cluster, and spreading the load in application logic to other nodes.

```
class YoutubeCrawlActor extends Actor with OculusSelections {
  val cluster = Cluster(context.system)

  val downloadersRouter = Router(RoundRobinRoutingLogic())

  override def preStart() = cluster.subscribe(self, classOf[MemberEvent])
  override def postStop() = cluster.unsubscribe(self)

  def receive = {
    case CrawlYoutubePage(url) =>
      val urls = extractVideoUrls(fetchPage(url))
      urls foreach { downloadersRouter ! DownloadFromYoutube(_) }

    case MemberUp(m) =>
      downloadersRouter addRoutee downloaderSelection(m)

    case MemberDown(m) =>
      downloadersRouter removeRoutee downloaderSelection(m)
  }
}
```

Using Akka's clustering module in the Loader allows the system to scale dynamically, without ever needing to be turned off – an important thing in long running distributed systems. The system turned out to be hard to measure accurately, due to the many moving parts (such as queue build up, non-determinism of which node would join at what time, and which video it would be assigned to download), yet generally speaking adding one more node would simply add one processing slot for the Download action – similarily as in the Hadoop scenario adding a node would.

Akka by itself does not provide simple tracking of message and it's results, which is inherently a hard problem to solve, since one would have to track each message's "parent". Akka is a rather low level tool, whereas Hadoop is a dedicated platform for tracking progress od Jobs in distributed systems – thus the conclusion is that it is easier to reason formally about a pure Actor systems behavior, than it is to strictly measure it (as least at the level of complication this problem is representing). Having this said, such monitoring would have to be built into the Oculus system, and is not provided by Akka itself – sadly this feature was not implemented in the reference system, and remains an open problem, as well as field of intensive research – as can be seen in multiple open source projects trying to solve this problem, such as Kamon [kam] or akka-tracing [akk].

Summarising Akka's scalability – as communication is done directly between two nodes after they have joined the Cluster, the overhead of communication is very low (as low as serializing a message and sending it over-the-wise). This also means that adding new nodes should have improved the system's performance in a nearly linear fashion, since the critical execution (slowest tasks) are executed in an "one Downloader per node" fashion, and these tasks consume the complete available memory and CPU limits given to JVMs running this application. Tracing the exact performance impact for such complex

message flows however, proved to be non trivial and has not been implemented as part of this thesis.

5.3. Summary of scaling methods

In this chapter we have seen multiple methods to scale distributed systems and encountered both gains and problems while doing so.

Both systems (the Loader, as well as the Analyser) were able to *scale-out* in a nearly linear fashion, although observing this exact change proved to be non trivial in the purely Akka based system. This is because Akka is rather meant to be as building block for systems such as Hadoop, and comparing them directly on this matter may seem unfair – as in comparing a tool to a car, built using tools. One should remember this when selecting to either ”use” or ”implement” an distributed computation platform, and take into account that using some solutions the monitoring already comes built in – as in the very old Hadoop eco-system – and sometimes it does not (yet), as is currently the case with pure akka applications.

The ease of scaling out both systems was really impressive, yet in the case of Hadoop one has to directly tell the master-node via configuration changes about the new slave node joining the cluster, as explained in Section 5.1.3 (”Growing the Cluster”). This is an inherent design flaw in Hadoop systems, since they always have one master node (although recent versions try to go away from this centralised topology). Akka on the other hand, with it’s *masterless cluster* is able to join nodes automatically, if they get in contact with any node that is already part of the cluster. Here we can see Akka’s clustering mechanisms being superior yet *less specialised* than the Hadoop one.

Summing up, both systems can be easily scaled up (by adding more powerful machines) or out (by adding more servers), and have displayed remarcable performance and stability throughout the tests conducted during this work. One should always remember that those two options should go in pair with each other, and that sometimes configuring the cluster in a better way may yield better results than blindy adding more nodes to the cluster. It is also crucial to have sophisticated monitoring installed in such cluster installations, as without them it would have been hard to detect and fix the cluster under-utilisation problem that was addressed in Section 5.1.4.

6. Conclusions

This chapter will summarise the findings from researching, developing and scaling the system implemented as part of this thesis.

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 stuff...

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

```
1  gcutil --service_version="v1" --project="oculus-hadoop"
2    addinstance "oculus-3"
3    --machine_type="n1-standard-1"
4    --zone="us-central1-a"
5    --tags="hadoop,datanode"
6    --disk="large-4,deviceName=large-4,mode=READ_WRITE"
7    --network="default"
8    --external_ip_address="ephemeral"
9    --service_account_scopes="https://www.googleapis.com/auth/..."
10   --image="https://www.googleapis.com/.../images/debian-7-wheezy-v20140408"
```

```
11  --persistent_boot_disk="true"
12  --auto_delete_boot_disk="false"
```

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

listing needs 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_list" that consists of names of cookbooks (which are simply a series of "steps to execute" in order to provision something) that should be applied to a given server, and then applying this "run_list" to a given server.

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

```
1  # knife solo prepare kmalawski@108.59.81.222 nodes/data-node.json
2  ...
3  (Reading database ... 42465 files and directories currently installed.)
```

```
4 Preparing to replace chef 11.8.2-1.debian.6.0.5 (using
      .../chef_11.12.2-1_amd64.deb) ...
5 Unpacking replacement chef ...
6 Setting up chef (11.12.2-1) ...
7
8 # knife solo cook kmalawski@108.59.81.222 nodes/data-node.json
9 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".

B. Bibliography

[akk]

[ble] Blender foundation website.

[Bru] Derek Bruening. bargraph – http://bargraphgen.googlecode.com/svn/trunk/bargraph.pl.

[CH73] Richard Steiger Carl Hewitt, Peter Bishop. A universal modular actor formalism for artificial intelligence, 1973.

[Com01] Creative Commons. Creative commons license – https://creativecommons.org, 2001.

[con] Concurrent inc. website.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, 2004.

[erl] Erlang programming language homepage – http://www.erlang.org/.

[ffm] ffmpeg project site.

[Fou08] Blender Foundation. Big buck bunny, movie, 2008.

[Goo] Google. *http://youtube.com*.

[Goo13] Google. Google's tesseract-ocr text recognition library, 1985 - 2013.

[had]

[Had12] Apache Hadoop. Hadoop sequence files format documentation, 2012.

[Inc13] Typesafe Inc. Akka remoting documentation, 2013.

[JB13] Victor Klang et al. Jonar Boner. Akka documentation, 2013.

[kam] Kamon – http://kamon.io.

[kmp] Fast pattern matching in strings.

[Ode13] Martin Odersky. Scala – http://scala-lang.org, 2003 – 2013.

[PH07] Martin Odersky Phillip Haller. Actors that unify threads and events, 2007.

[RSB77] J. Strother Moore Robert S. Boyer. A fast string searching algorithm. 1977.

[sca]

[Wik] Wikipedia. Column-oriented database.

[Zau10] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions, 2010.