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

. . . . . .

PODPIS

I would like to thank Dr Ernst, my wife, family and friends for all the support given to me during the creation of this thesis.

Contents

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 a 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 scaling out big-data systems, which have been acquired 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 deals with simple image processing algorithms and near-duplicate detection of movies, or scenes within movies.

For this thesis's brevity, the focus will be on video material, of which the amounts of freely and legally available materials are significant – especially 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 commonplace in today's world, where everything is recorded, published, possibly modified and distributed again, all 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 challenging task of efficiently handling this data. Tasks such as de-duplicating, searching, categorising or extracting features (e.g. text) are now even more challenging and exciting than ever before. Together with the staggering amounts of data these systems have to deal with, this new range of applications poses a significant challenge and interesting opportunity to develop new kinds of paradigms as well as algorithms, geared specifically 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 referred to as *”Oculus – the video material analysis platform”*, will be tasked with processing enormous 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 infringement can be avoided.

The primary goal for this application is to expose and highlight challenges 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 experience 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 the reference system needs to solve.

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 [You] 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 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 ourselves 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 convenience 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 profits, 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 in Section 4.2.

1.4. Thesis structure

Chapter 1 serves as broad overview and introduction into the problem area of this thesis. It also introduces the need for a reference implementation on top of which recommendations are made in latter chapters of the thesis. Lastly, a number of goals are set before 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 and 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 optimisations 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 serves as significant data point in determining whether or not the selected technologies are in fact scalable or not.

Chapter 6 summarises recommendations collected during implementing and tuning the system as if it was an in-production running system. The section retrospects about the system’s scalability and appliance feasibility in real-world systems. Lastly it reiterates the thesis’s goals and judges whether it was possible to fulfil them.

2. Analysis of available technologies

The primary focus of this work is analysing and benchmarking usage of popular distributed system stacks. This section introduces the various technologies used throughout the work on this thesis.

This chapter should be treated as a brief introduction into the selected technologies, as in-depth 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 modelled after a number of Google's most famous whitepapers concerning "Big Data", such as *Chubby* [MB06] (on which the *Google Distributed File System* [SGSTL03] was built) and the later, 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 originally an implementation used by Yahoo [Yah07] internally, and then released to the general public in late 2007 under the Apache Free Software License.

The general use-case of Hadoop based systems 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*" or "emit") and *reduce* (also known as "*fold*") functions in a parallel, distributed fashion. Complex computations, which can not be represented as single Map Reduce job, 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 them. These claims will be investigates in Chapter 5, where results of different cluster configurations will be compared. It's computation model proposed is examined and explained in detail in later sections of this paper, as it is the main model chosen for the implementation of the presented system.

2.2. Scalding & Cascading

Scalding [sca] is a Domain Specific Language implemented using the Scala [Ode14] programming language. It has been developed at Twitter [Twi14] 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 [JG95], which would often result in very verbose and hard to understand code (especially 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 [con13] Cascading library, which is a framework built on top of Apache Hadoop. It provides 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 [col] designed in accordance to the Google "BigTable" whitepaper [FCG06], describing their datastore implementation published in 2007.

Column-oriented storage of data, as opposed to row-oriented (as most SQL databases), has huge advantages when many aggregations are performed over only a subset of the columns – as loading the other columns into memory can be avoided. Because of this specific feature of HBase it is very common o have rows that span thousands of columns, while still remaining efficient.

It was selected for this project because it's excellent random-access to data as well as for 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 of it's Tables are laid out as Sequence Files that are a very performant way to store data, such as rows of a big table, on HDFS. An in-depth 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 [Ode14] running on the Java Virtual Machine. It was selected as primary implementation language for this project due to Akka and Scalding libraries. Both libraries are introduced in this section, both allow building distributed systems. Other languages used during this thesis include: ANSI C, Ruby and Bash, yet the vast majority was implemented using Scala.

Scala'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 [Typ13b] is a library providing actor model [CH73] based concurrency utilities for Scala and Java applications. Using this concurrency model can be best explained using two rules "share nothing" and "communicate via messages" – both stemming from the programming language Erlang [erl]. 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 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.

During the work on 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 hash, yielding „how similar" these hashes are. The details and implementation of it have been explained by Christoph Zauner [Zau10].

This algorithm is used by the system to perform initial similarity analysis between images. It is publicly available, including it's C source code, and may be used in *non-commercial applications* in accordance to it's license. 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 deemed appropriate to use an existing perceptual hashing solution.

2.7. Chef

OpsCode Chef [Che13] is a set of tools aimed at automating server configuration and provisioning. Using it enabled automating spinning up new servers which was a very large part of the work on this thesis. Once such automation was prepared, it could be applied to different cloud service provides – which led to exploring local virtual machines[1] as well as Amazon's Elastic Compute Cloud (widely known as "EC2") public cloud offering until lastly settling on using Google's Compute Engine public cloud offering. Chef prevented spending many tedious hour of reinstalling the cluster each time anew. Chef allowed to "cook" given virtual machines into the required state – all software that is required to run the application was installed by one-written chef scripts instead.

Appendix A features an in-depth 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. Youtube-dl

Youtube-dl [Gon14] 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. It is aware of multiple available video formats (high / low quality). It offers multiple options useful yet hard to implement for this project.

[1] Vagrant, which utilises VirtualBox

Another reason for choosing youtube-dl lies in useful yet hard to implement options it offers. Prime example of such option would be possibility to "prefer downloading open source video formats. Thanks to this option the youtube crawler system was able to download only free formats, for which conversion utilities are freely available on GNU platforms.

3. System design

The system is designed with both: asynchronous and distributed approaches in mind. In order to achieve high asynchronicity 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** – responsible for obtaining more and more reference material. It persists and initially processes the videos, as well as any related metadata. In a the real world, such reference data is usually provided by partners, such as movie or television companies.

– **Analyser** – 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.

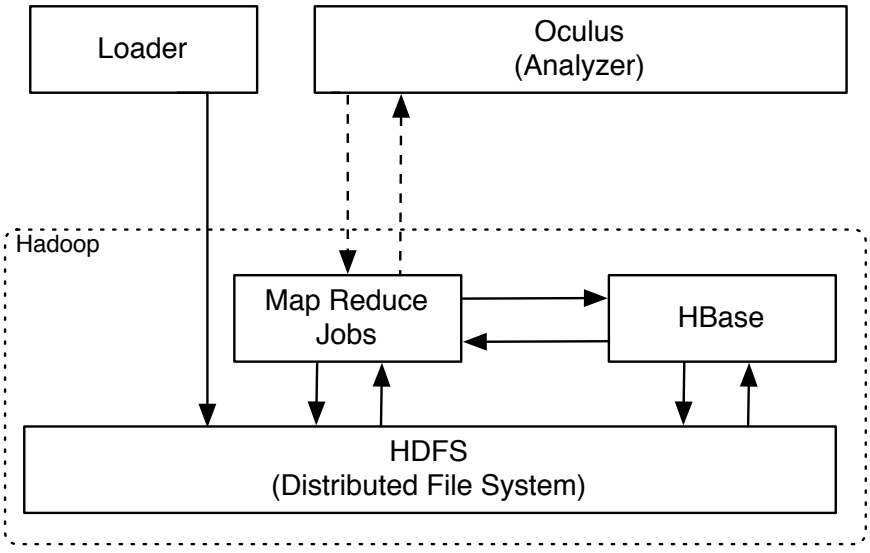


Figure 3.1: High level overview of the system’s architecture

This section will focus on describing the general design of both systems, first the Loader’s actor-based cluster model, and then the Analyser’s Hadoop-backed batch processing cluster.

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 or 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 *Cluster* module [akk], it was also fairly easy to distribute actors among multiple physical nodes in the cluster, allowing the actor system to utilise the entire cluster's processing power. The use of messaging as default and only mean of communication with actors allowed to easily transition into the clustered model, where communication would he handled over the network, from a previously local-only model.

3.1.1. Types of Actors used in the system

The system consists of 4 types of Actors. Each of them 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 explaining the protocols used for communication between them in the running system. These are very important as actors can communicate only via sending and receiving messages, and can not provide any type signatures that would help to identify what messages they can respond to. The also do not have methods that can be called directly on them – this is a direct result of implementing actor model based concurrency, which would not be able to hold in face of direct access to an actors state.

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 and crawling of "related video sites" (`Crawl(siteId: String)`). It also sends messages that trigger the download process by sending `Download(movieId: String)` messages.

  receives:

  `Crawl(siteId: String)` messages

  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** – downloads 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.

– **ConversionActor** – converts the downloaded video material into raw frame data (bitmaps).

  receives:

  `Convert(localVideoFile: java.util.File)` – which 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** – stores 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 messages 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 (node-01, node-02, node-N) of the cluster. Figure 3.2 together with the flow description in this section aim to provide the necessary high-level overview of the system’s execution flow.

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

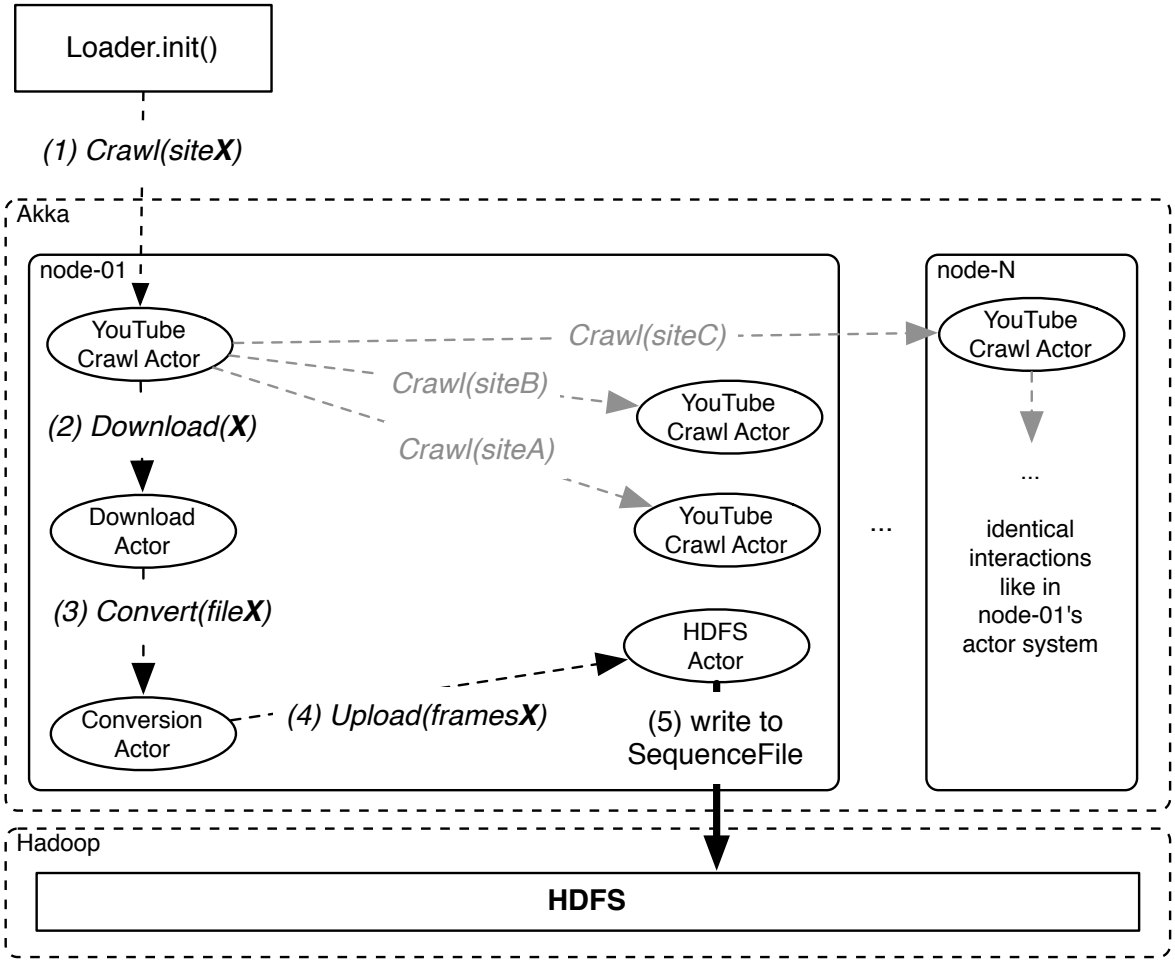


Figure 3.2: Overview of messages passed within the Loader's actor system. Greyed out messages are also sent, but are not on the critical path leading to obtaining material from *siteX* into HDFS.

of Routees (target Actors), and decides to deliver a message only to the Actor who has the smallest amount of messages "not yet processed" (which are kept in an Queue called the "Inbox", hence the strategy's name).

Step 2 - *Download* messages

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

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

Step 3 - *Convert* messages

The next step is executed by an instance of an *ConversionActor* receiving 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 un-compression 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 enabled), 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, restricting 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* receiving 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 to around 2MB per frame (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 Analyser 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 a fully asynchronous system, composed of Actors performing very specific tasks. All communication between the Actors is implemented as message passing, which allows for *location transparency* between the Actors – this property has been utilised to spread the work-load between multiple nodes in a clustered environment, enabling 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 Actor 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.

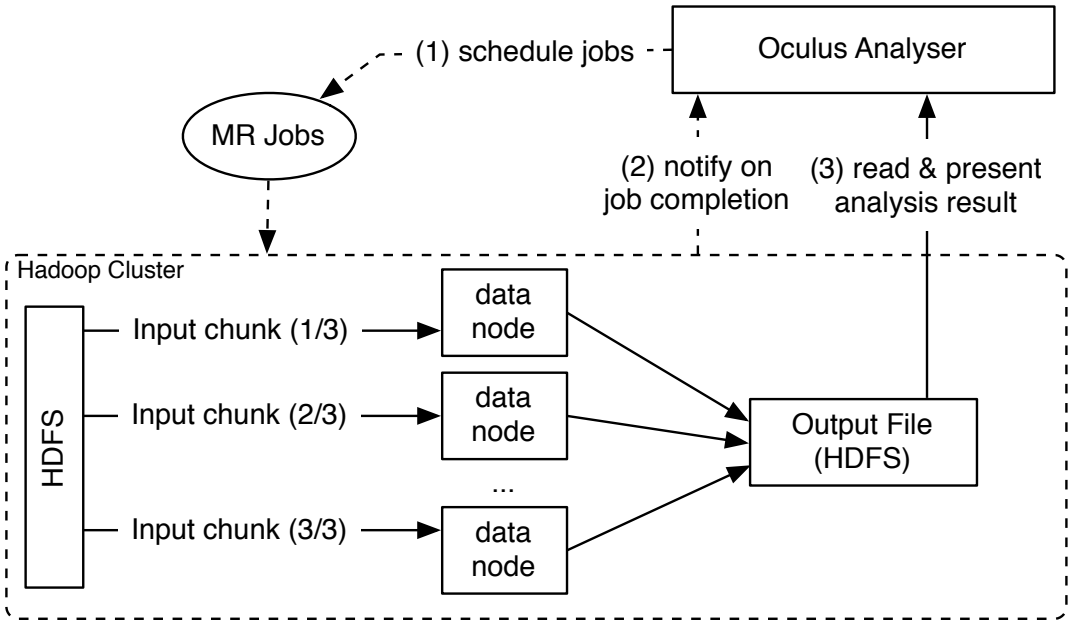


Figure 3.3: Overview of the Analyser's execution flow, when working with

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 by launching another analysis job based on the notification that one pipeline has completed. This allows the system to keep the 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).

The following sections (3.2.1, 3.2.2) will introduce the cluster deployment as well as how Jobs are implemented on top of it, in order to parallelise the job's execution as much as possible.

3.2.1. Hadoop Cluster deployment strategy

Since the Analyser is most dependent on the Hadoop cluster's deployment this section aims to introduce the cluster's components and initial (3 node) setup.

While this section only only introduces the various components of the cluster and it's initial deployment, chapter 5 ("Cluster scaling and performance analysis") revisits this topic and will focus on details such as cluster configuration options as well as influence of the number of participating nodes in the cluster on total computation times will be measured and tuned to increase the cluster's efficiency.

Initial Hadoop cluster deployment

The Hadoop deployment described in this subsection will focus on two elements of the Hadoop ecosystem - the Hadoop Distributed File system (HDFS) and the Hadoop Map Reduce Runtime.

The minimal hadoop deployment consists of *5 individual processes*, each with an assigned and very specific role in the clusters functioning. These processes can be classified as running along side the NameNode or DataNodes. This distinction should be clear after listing the names and features of these processes.

There are 3 processes associated with the "master node", namely: the NameNode, the SecondaryNameNode and the JobTracker which is responsible for coordination of HDFS operations and Map Reduce Job scheduling:

- **NameNode** (HDFS) – responsible for mapping logical file paths to their locations on DataNodes within the cluster. This component was a Single Point of Failure (SPOF) on Hadoop clusters until version 2.0 implemented the so called High Availability (HA) module, which allows the cluster to run a second (not to be confused with *secondary*) NameNode in the cluster in the case the primary would fail. The HA module was not used during this thesis, as it was not required to ensure the cluster's high availability as the data could always be easily obtained again.

- **SecondaryNameNode** (HDFS) – responsible for a process called "checkpointing" of the NameNode's fsimage and it's edit-log. It's name may be very misleading, as at no point in time is the SecondaryNameNode able to take the primary NameNode's place – this can be only achieved by using the High Availability modules. Use of the SecondaryNameNode though allows the NameNode to fail and re-join a running cluster – without the SecondaryNameNode that failed NameNode would have lost all of it's file system state and a full HDFS restart (including all datanodes) would be required.

- **JobTracker** (MapReduce) – responsible for scheduling Map Reduce Jobs running in the cluster. It behaves like a queue, taking in Job requests and processes them in order (or based on customisable strategies). It is also in contact with all TaskTrackers (described below), which actually perform the work related to a MR Job, while the JobTracker only gets and aggregates the job's progress. In case of failed Jobs or Tasks it can also restart or redistribute the job to other DataNodes.

Secondly, the cluster contains 2 types of processes associated with *slave* nodes. They are mostly responsible for "the actual work", whereas the processes running on the *master* node can be seen as "coordinators". Each salve node in the cluster is running these processes:

– **DataNode** (HDFS) – responsible for the actual storage of data on HDFS. Data nodes themselves are not able to correlate a path to a given file data block – only the NameNode is able to perform this lookup, which also means that in the case of the NameNode failing, the data stored on the DataNodes is temporarily unavailable. The DataNodes can talk to each other in order to migrate and replicate the data blocks directly with each other, though this action has to be initialised by the NameNode.

– **TaskTracker** (MapReduce) – responsible for scheduling and executing Jobs on a given DataNode. Tasks are chunks of work, issued by the JobTracker to TaskTrackers, which then handle their execution. Once all tasks for a given job are completed, the JobTracker reacts by completing the overall job. A TaskTracker should be run on the same physical node as the DataNode, because then it can leverage this fact for node-local task execution.

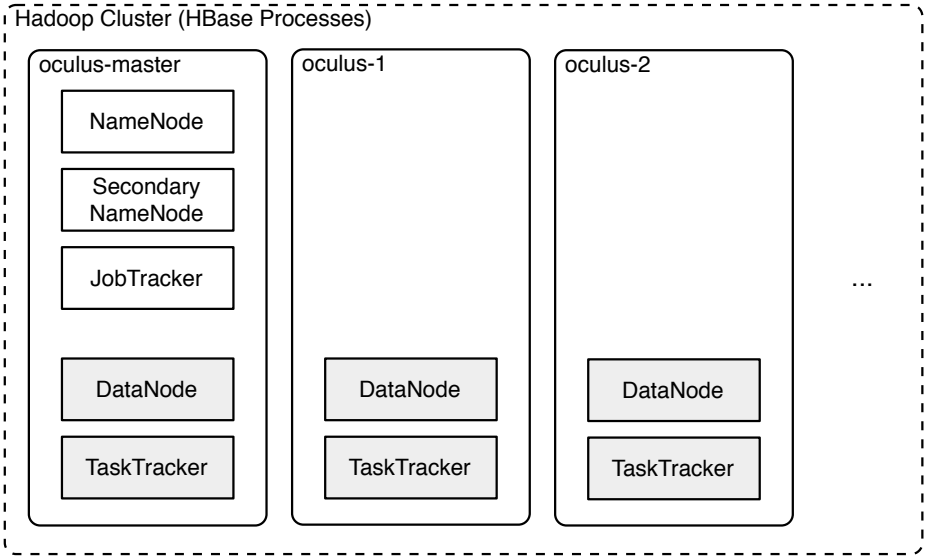


Figure 3.4: Deployment diagram of the various HDFS processes within the initial 3-node cluster.

These 5 types of processes must be running in the cluster for Map Reduce jobs to be executable by the cluster. All nodes must be able to communicate with each other. Firstly, because the master node must be able to submit tasks to it's slave nodes (DataNodes) and secondly, because DataNodes must be able to communicate between themselves in order to replicate data among the cluster. Starting the cluster however does not require the administrator to start these processes on each of the machines separately – as the master node is able to issue start/stop commands via `ssh` to it's slave nodes. This also means that the slave nodes do not need to know about all their "neighbours", because the master node will inform the newly joined nodes about the cluster's current state and IP addresses of the remaining DataNodes.

HBase deployment

Discussing the Hadoop cluster's deployment actually automatically covers part of the deployment information needed for the HBase cluster. The reason for this being, that the HBase cluster's equivalent of "data node", known as RegionServers must reside on the same physical nodes as HDFS's DataNodes.

HBase also has a clear separation between processes that are *master* processes and *slave* processes. The master and coordination processes are:

– **HMaster** (HBase) – responsible for all coordination of reads and writes to HBase. Although it may seem that it might quickly become a bottleneck, since all client applications connect directly to it, this is not the case. While clients indeed do connect directly to the HMaster all read/write operations are performed directly between the client and specific RegionServers which host the data that a given client query is touching.

– **HQuorumPeer** (ZooKeeper) – is HBase's wrapper around Apache ZooKeeper [Fou13], which is a centralised multi-node quorum-based metadata storage service. ZooKeeper is used for cluster management within HBase, and the HQuorumPeer serves as "managed by HBase ZooKeeper cluster". Thanks to HB running the ZK cluster by itself, for small clusters (less than hundreds of nodes), it eases the administrative overhead of managing another cluster manually. ZK is used to register and auto-discover RegionServers, once they notify ZooKeeper of their existence.

While, exactly as with the NameNode in the case of HDFS, the HMaster should be a cluster-wide singleton, and serves as entry point for client applications for issuing reads and writes, the HQuorumPeer has to be run on multiple nodes.

In initial configuration of the HBase cluster the HQuorumPeer was run on three nodes in the cluster. The reason for using 3 nodes to run the HQuorumPeer (ZooKeeper) processes is that, since ZK operates on a quorum it requires at least `ceil(N/2)` nodes to be available to make progress. In an 3 node scenario, this means that the cluster will continue to work properly while at least `ceil(3/2) = 2` nodes are available. In other words, setting up 3 nodes to run ZK, allows 1 node to fail, without stoping the entire cluster.

Continuing the discussion about process types running in the HBase cluster, the slave nodes require one type of component to be running:

– **HRegionServer** (HBase) – responsible for processing client requests to read / write to HTables. The name "Region" mentioned in it's name refers to the fact that the *row-key* space of a Table is split among multiple servers, so that when performing a full-scan filtering operation, multiple servers can perform the scanning *in parallel* – each for it's own section of the *row-key* space. It must be running along side with a DataNode on the same physical server.

While the slave type of process is much simpler for HBase than for HDFS, it does have one additional requirement: an HRegionServer must be running on the same physical node as a HDFS DataNode. The reason behind this is that RegionServers store their assigned region's data on HDFS and by running it on the same node as a DataNode, all writes can be made locally – without transferring the data over the network. Of course, if HDFS is configured to replicate it's data to multiple nodes – it will, but this does not concern the HRegionServer.

Having the HRegionServers deployed on the same nodes as DataNodes has another effect: all optimisations performed to HDFS and the DataNodes, will directly influence HBase's behaviour. One should also keep in mind that some features may seem to be duplicated among HDFS and HBase – such as replication, which can be set up directly on HBase or HDFS. Configuring some of these settings in HBase

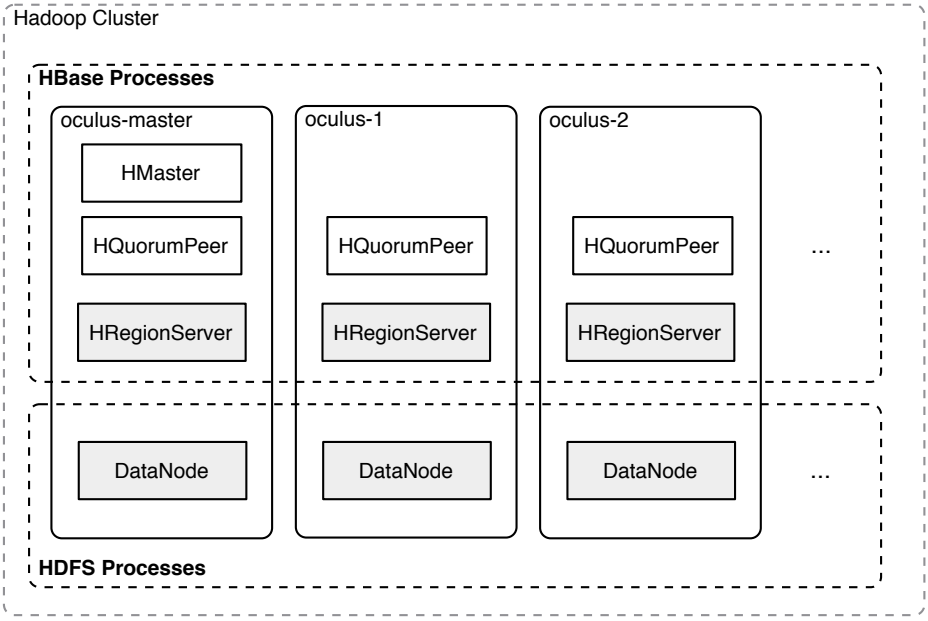


Figure 3.5: Deployment diagram of the various HBase processes within the initial 3-node cluster. DataNodes are not directly part of the HBase cluster, yet are required to be running on the same physical nodes as HRegionServers.

rather than HDFS has the benefit of HBase being more aware of the structure of the data it is storing, whereas the same data carries nearly no meta information HDFS can use to redistribute it more efficiently. Namely, HBase can redistribute data related to Regions of HTables more efficiently, so that full scans on these tables can be parallelised more evenly and predictably.

All diagrams in this Section have included an "$\cdots$" marker, hinting at the possibility to add more nodes to the cluster. When scaling Hadoop clusters the only type of node that has to be added is the "slave" kind of node, which have been positioned right-most on each diagram. In fact, chapter Chapter 5 will focus directly on scaling out these clusters, with one of the tests scaling it out to 10 nodes.

3.2.2. Defining Map Reduce Pipelines using Scalding and Cascading

The language and for implementing these processing pipelines used in this thesis was Scalding [sca], which was already briefly introduced in Section 2.2. The choice of Scalding, and not Hadoop's plain Map Reduce API, as a way of interacting with Hadoop jobs from Scala, strongly influenced by the increased 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 on top of Hadoop yields, by showing the smallest possible example of an Map Reduce Job, implemented using both plain Hadoop APIs and Scalding. One should remember that Scalding is not a different framework to Hadoop, and in the end it will produce the exact same computations and classes (one instance of a `Mapper` and one instance of a `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 occurrences 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 tuples together by the first value (the `word`), and reducing these into a tuple of `(word, sum(1,1,1,...))`.

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 boilerplate 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.

Such pipeline can be visualised as seen on Figure 3.6, which is an output generated by using the graphviz tool, applied to a file describing the graph as described in the DOT format. The DOT file corresponding to the graph depicted on Figure 3.6 is shown in Listing 3.4. DOT descriptions of pipelines are in common use among professionals designing such systems, and can also be generated automatically by Scalding – which is tremendously useful when working with very long pipelines.

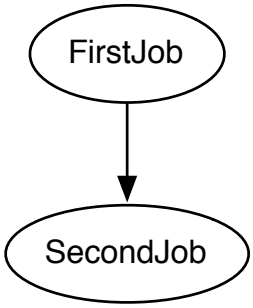


Figure 3.6: Simplest Job Pipeline, with SecondJob depending on the output of FirstJob (each circle being one Map Reduce Job).

Listing 3.4: Textual description of graph on Figure 3.6, 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 of Scalding's rich set of operations, what may look as simple on the surface (for example, 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 =
    WritableSequenceFile(input, ("key", "value"))
    .read
    .rename("key", "frame")
    .map(("frame", "value") -> ("dominating", "red", "green", "blue")) {
       p: (Int, BytesWritable) => calculateHistograms(p._2)
    }

  // reference database
  val referenceFrames =
    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. It reads from 2 separate data sources, one for the reference data, and one for the currently analysed video. The data for the analysed movie is taken directly from the analysed SequenceFile, containing the frame data. The data of the reference database is read from the `HistogramsTable` which is stored in HBase. The `HistogramsTable` refers to an *HBase Table*, and during the Job's execution 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 modelled by this definition.

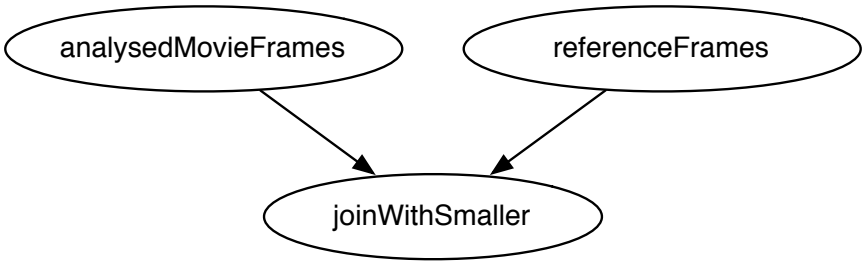


Figure 3.7: Join operation, forcing the Pipeline to consist of 3 Map Reduce Jobs (depicted as circles)

The dependencies, as seen on Figure **??** between the first two Jobs to the 3rd "Join Job" were not explicitly modelled as previously seen using the `next` method, and instead were inferred from the use of the `job1.joinWithSmaller(job2, "key1" -> "key2"` operation. This is very important and useful since:

- Cascading is able to determine dependencies between Map Reduce jobs, and execute them on the Hadoop cluster as dictated by their dependency graph.
- Jobs that do not depend on each other's outputs can be run in parallel.

Thanks to the second property of these Job Pipelines (parallel execution of independent sub-graphs) total execution time of such Map Reduce Pipelines may be further optimised when running on large clusters.

3.3. Analyser design summary

Summing up the Loader's design, it should be clear that most of the architecture required for it's operation is in fact a classical Hadoop cluster. The application itself does not need any additional opera-

tional servers, as the only thing the Analyser does is to issue compile Scalding Job descriptions and then send these to the cluster's JobTracker, which takes care of their scheduling and execution.

The Analyser should be seen as an end-user application. It is run locally on a developers work-station in order to prepare and issue various jobs. This has obvious gains in terms of not requiring the developer's machine to store these humongous amounts of data, and only issue precise queries and jobs.

4. Analysis Job implementation examples

This chapter aims to provide tangible examples of how the previously described system works and in what way the parallel nature of the implemented system benefited these use cases.

The following two sections will focus on practical examples of so-called ”attacks” (distortions) 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 analysed in order to find it’s corresponding ”mirrored” counterpart. 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 speed up the system’s 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 search methods similar, in concept, to sub-string search could not have been applied efficiently. The section explains the algorithms applied instead, and touches upon the problem of interpreting results of complex map reduce jobs.

Section 4.3 summarises how the approach to designing pipelines and data representation used in this chapter was influenced by the the large amounts of ”fuzzy” data, and how these algorithms could be further expanded.

As this thesis is not focused on development of image analysis algorithms, these sections will instead lean their focus towards the distributed system and big-data parts of the problem. The image comparison algorithms also assume that that the used image correlation algorithm (phash – see Section 2.6) performs well enough for the job at hand.

4.1. Near-duplicate video detection

In order to test the system in scenarios of image distortion the example case of ”mirrored” video material was first used. This case is fairly popular among material uploaded to YouTube, as content uploaders often ”flip” the uploaded content in order to avoid copyright detection to trigger on content.

As will be shown in the example section, this ”attack” is not very effective, as determining a exact-mirror video material is fairly simple. Even in the case of slight hue and luminescence changes the implemented system was able to detect such videos flawlessly.

4.1.1. Analysed example material

The video material used in this chapter to exemplify the described attack scenarios originates from the movie "Big Buck Bunny" [Fou08] which has been created by the Blender Foundation [ble] and released under the Creative Commons license [Com01].

Figure 4.1 for example, illustrates the original as well as mirrored counter part of an example frame taken from the movie. The movie is 9 minutes and 59 seconds long. The version used during this analysis is the original 1080p resolution version, recorded by the Blender foundation. The entire movie was stored in the reference database in raw format, which amounts amounts to a total size of 6.25 GB.

(a) Original frame (b) Mirrored frame

Figure 4.1: Example of original and mirrored frame

4.1.2. Histogram calculation and HBase row key design

The Map Reduce pipeline designed for detecting near-duplicate movies takes one very important fact into account: the histogram of an nearly identical movie will most likely not change very much. In the case of merely mirrored video material, the histogram will not change at all. This fact was utilised to design the row key for storing histograms in HBase for efficient lookup.

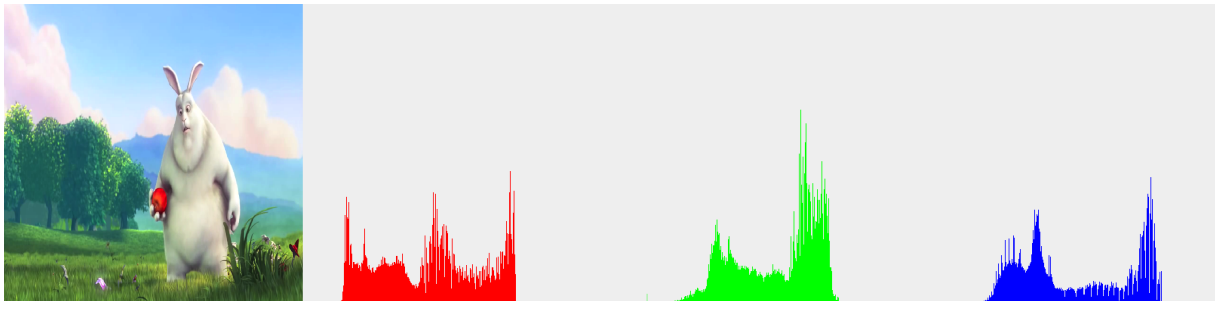


Figure 4.2: R/G/B Histograms of analysed frame

Hbase's implementation provides strict guarantees about an HTables row key, specifically it guarantees to store data in *alphabetically sorted order* based on it's row key. This property *must* be taken into account when designing tables and keys in HBase. In particular it is a widely used technique to store parts of the data, or projections of it, directly in the key itself – this is because HBase relies so much on the key to distribute data among RegionServers and a wrongly designed key may easily hot-spot one of the region servers (even in the presence of so called region-splitting, which re-shards the more work-intense regions onto multiple servers).

In our example, after obtaining the histogram data for each of the frames of an analysed movie, it is stored in the `Histograms` table, with a precisely defined key. The key is designed to allow *prefix queries* which are incredibly fast – because HBase is able to determine, just by looking at the prefix query range, which exact RegionServers must be contacted, and they in turn are able to perform a linear scan of just the required region of the Table (which often means reading from disk). The key was designed using these guiding principles:

– the key must allow for prefix queries, which given a ”to be compared frame” will return ”probable candidates”

– more precise queries must also be possible to be constructed as prefix queries

Having this in mind, the key is designed to be an descending sorted − joined list of colour dominance in a given frame. Colour dominance is calculated based on the percent of pixels in the image where one colour's part would dominate (have a higher value) than any other colour parts. For example given a pixel `RGB("#F2649D") = RGB(100, 242, 157)`, this pixel would be deemed to be *green dominating*.

Listing 4.1: Grammar for the Histogram RowKey. In essence, it contains an encoded colour dominance value, the youtube id of the movie, and frame number.

```
1   NUMBER       : [0-9]                        ;
2   YT_ID        : .+                           ; // youtube id
3   FRAME        : NUMBER+                      ; // frame number
4   COLOR        : ("R" | "G" | "B") NUMBER  ; // dominance of color in frame
5   HIST_ORDERED : COLOR "-" COLOR "-" COLOR ; // sorted by dominance
6
7   ROW_KEY      : HIST_ORDERED ":" YT_ID ":" FRAME ;
8
9   // examples: R57-B28-G13:YE7VzlLtp-4.mp4:22009 <- Red dominates
10  //           G47-R41-G11:YE7VzlLtp-4.m6p4:919 <- Green dominates
```

Listing 4.1 provides a simple ANTLR [1] inspired grammar, which should make the key's parts easily visible (while ANTLR was not used directly for parsing of the key, it is a very clear and concise way of explaining the key's structure).

Having defined such key, Map Reduce jobs populate the reference database using it. Of course, the entire histograms would still be stored within the row (all 255 values for each color). The rest of the HTable's design is depicted on Figure 4.3.

This design allows the Analyser to issue very efficient queries for frames, such as in the examples shown in Table 4.1. For one matching job, the analyser will issue multiple HBase queries, broadening the scope of the search if no full match has been found in the first round, avoiding having to issue slower Scan operations to HBase.

[1] ANTLR is a popular parser generator library for Java applications

	hist:			phash:		youtube:
RowKey	red	green	blue	dct	mh	json
...	167,...	163,...	203,...	...	...	...
G40-B35-R25:YID:F####	203,...	124,...	123,...	...	...	...
G40-B39-R19:YID:F####	211,...	11,...	14,...	...	...	...
G41-B39-R18:YID:F####	124,...	224,...	221,...	...	...	...
...	...	...	...	...	...	...
R70-G17-B13:YID:F####	122,...	53,...	33,...	...	...	...

Figure 4.3: HBase HTable design used for histogram based lookups.

Question	Query	Prefix Scan
Blue dominated frames	`START "B", END "B99"`	Yes; Fast!
Strongly red dominated frames	`START "R9", END "R99"`	Yes; Fast!
Green and Blue dominated frames (grass + sky)	`FILTER "G*B*"`	No; Slow
All frames of Movie	`FILTER "*-e98uKex-*"`	No; Slow

Table 4.1: Examples of queries enabled by such RowKey design

4.1.3. Role of perceptual hashing in near-duplicate detection

Since simple histogram comparison if obviously not enough to determine if an image is of the same movie or not, the second layer of comparison employed by the Analyser's Map Reduce Jobs is powered by perceptual hashing, as implemented by the phash [Zau10] library.

As can be seen in Figure 4.3, rows in the Histograms table contain not only the histogram values, but also the phash column family as well as youtube metadata. In this step of execution we will be comparing the already pre-filtered (by using the previous method) frames by their perceptual hashes. A detailed description and effectiveness analysis of these is available on the libraries website [Zau10].

During this work two perceptual hash implementations were used:

– DCT hash – Discrete Cosine Transform Hash; Is an efficient means to calculate an image hash based on it's frequency spectrum data. It is usually not sufficient in order to determine image similarity, but is a good accompanying hash function, which can be used in pair with other functions to increase their accuracy. If DCT does not match the given images, then most probably other algorithms also should not. It is also fairly efficient in detecting slight rotation and blurring. Experimentation conducted during this thesis show that a distance value lower than 20 points usually indicates a "possible match" using this algorithm.

– MH hash – based on the Marr–Hildreth algorithm [mar80], which is an edge detection algorithm. The resulting "hash value" is in fact an compressed representation of the edges detected within the analysed image. This hash has a constant length of 72 bytes long hash, and can be compared efficiently by using the classical Hamming Distance [ham] method. This hashing algorithm is

usually sufficient to determing image equality. It is not very resilient against image rotation, but manages very well with blurring and differences in compared image resolutions. Used in pair with the DCT hash and histogram methods it is a very viable approach to near-equality checks.

Listing 4.2 presets example hashes calculated for the frames seen in Figure 4.1. It's worth pointing out that the dct hash in both cases is equal, while the mh hash is different changed - because it carries more information in it's 72 bytes.

Listing 4.2: Example hashes, calculated for a original and mirrored frame

```
1  // Big_Buck_Bunny_normal.png
2  dctHash = 54086765383280
3  mhHash = 8e2f6d04831568425b3c5c2ca01afd88b6ad638d43ced55d71cc032c53a
4           bdc898e930523e00fb334e765e57529f0ca5b6b6333a35076b1fd249a1e
5           00e0e7c45beaa10792bc453228
6
7  // Big_Buck_Bunny_mirror.png
8  dctHash = 54086765383280
9  mhHash = 458193aa7ca2e2f03e29065f29f01019f4b813f07b217f94775dcb91985
10          cc0ef83656b981b575f4d44ad7848c5469d1ad81a496665c3e262070e42
11          4d8592231a1c8118fcf2f0ef63
```

Hashes like this can be compared to each other using a simple Hamming Distance [ham] function, as their values at certain positions represent certain traits of the hashed picture. This is the core of the last phase of the matching algorithm. The Table 4.2 shows how the distance between these hashes differ in a few example scenarios. As is to be expected, the mh hash is reacting much more than the plain DCT hash.

Tested distortion	DCT distance	MH distance
Mirrored frame	0	402
+20% luminosity	0	132
Blurring	0	246
Slightly different frames (next frame)	25	324
Different frames (moved character)	31	355

Table 4.2: Example distances between sample frames, and their distorted versions.

After a certain number of experimental runs with the system, it was empirically decided that images whose distance values ranged below 40 for DCT-hash *and* very near or below 350 for the MH-hash can be *safely* considered "almost identical".

Results from such a movie comparing job pipeline are yielded onto HDFS, in form of a simple CSV file, which is sorted by multiple fields: first the input frame, then the histogram distance and then the hash distances. Using this method it's possible to determine "top N" most probable matches for a given frame. Listing 4.3 shows good top matches – such the dct distance is below 40 and the mh distance below 370. In this case, since the analysed movie was a mirror the histogram difference (calculated in a very simplistic way – as hamming distance of the row-key encoded histogram data) is equal 0 for identical

(just mirrored) frames, and is slightly off for cases where the reference database did not contain the exact now matched against frame. Since the reference data is not stored for every frame for movies, but for 10 frames per second (instead of the usual 25 in PAL or 30 in NTSC regions), the granularity of the matches will be down sampled to a granularity of 10 frames. This lower data granularity is the reason that some frames will match a very near frame in the reference database. This storage-space optimisation could be discarded if high quality matches would be required by the system.

Listing 4.3: Examples of valid ”best” matches for a few sample frames (mirrored movie).

```
...
F  7809 => dist(dct: 38, hist: 3, mh: 376) => YE7VzlLtp-4 @ 7789 (20 off)
F 14409 => dist(dct: 34, hist: 0, mh: 349) => YE7VzlLtp-4 @ 14409 ( 0 off)
F  2599 => dist(dct: 37, hist: 2, mh: 351) => YE7VzlLtp-4 @ 2569 (30 off)
F 54919 => dist(dct: 32, hist: 0, mh: 307) => YE7VzlLtp-4 @ 54919 (0 off)
...
```

Listing 4.4 on the other hand, shows example values for an invalid matches. All distances are unreasonably high – especially such high value of the dct hash distance disqualifies these frame matches from further processing.

Listing 4.4: Examples of invalid matches for a few sample frames.

```
...
F 4299 => dist(dct: 53, hist: 3, mh: 395) => YE7VzlLtp-4 @ 57809
F 7539 => dist(dct: 47, hist: 2, mh: 384) => YE7VzlLtp-4 @ 29939
...
```

Movie matching pipeline summary

To sum up how the pipeline was designed, and how the previously described parts fit into the bigger picture Figure 4.4 explains how the different operations are connected into the pipeline.

Describing the Pipeline depicted on Figure 4.4 in further detail: First the attacked movie material must be hashed as well as it’s histograms must be calculated (1). This data is then persisted into HBase. The Join operation (3) forces the HBase read (2) and performs the joining operation. The now merged data stream is then used to calculate the hash and histogram distances between all (millions) joined frames (4). Next, the data is grouped by the compared movie’s frame numbers – this allows us to find the best match for each frame, by simply sorting the sequence of data grouped for each of the frames (5). Scalding allows to perform this operation quite optimally – so that not the entire stream must be sorted, we simply find the best value and stop sorting. Next the data is again grouped, but this time against the matched movie ids (6), which allows us to count how many frames matched to a given movie. Lastly, simply sorting the output of this Job by the number of matched frames (7) allows to find a movie that "the most frames have been matched to".

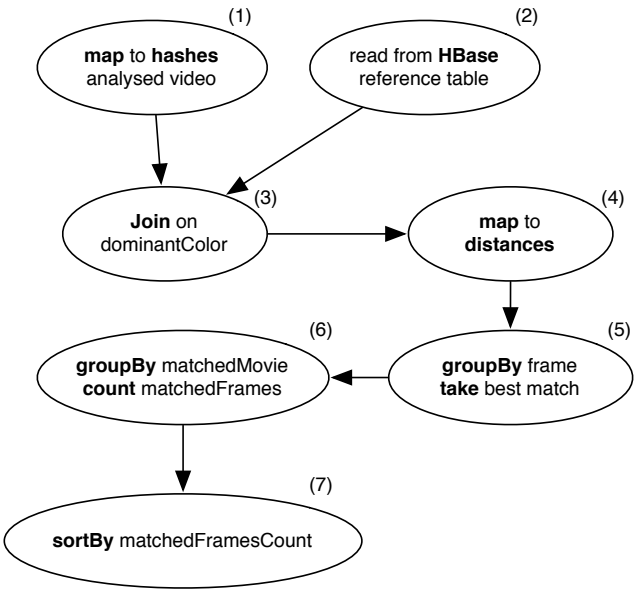


Figure 4.4: Map Reduce Pipeline, designed for matching ”most similar” movie in reference database, when an attacked video file is given.

4.2. Scene positioning

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.

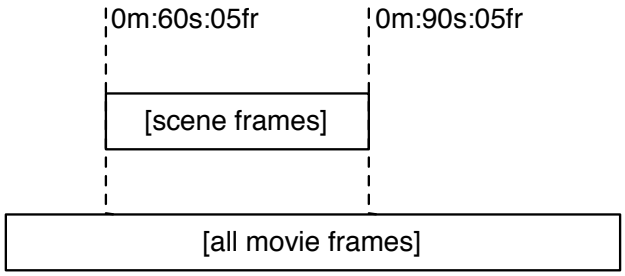


Figure 4.5: Visual representation of the goal of this example application.

Although on the surface 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 [DEK77] 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 distortions in source and reference video material.

Additionally the possibility of cross-matches when a movie is built up from multiple short scenes from other movies. The most popular example would be 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 distortions is frame rates of reference data vs. an analysed video fragment – even

a slight mismatch (30FPS vs. 25FPS) would render the substring search algorithms not usable for this concrete example.

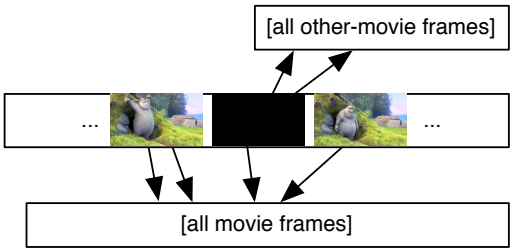


Figure 4.6: A frame may match multiple reference movies, complicating the task significantly.

Instead, a more statistical approach was taken. The basic idea of this Pipeline can be described as trying to match frames between 2 movies in similar style as in the previously described pipeline (Figure 4.4, yet afterwards the top match for each frame must proceed incrementally in the matched frames. For example if scene frame 100 matched reference frame 90100 and scene-frame 110 matched frame 90110 one should expect that this interval will be constant over the entire matched range of the frames. This method can also detect ”top10” movies, where a movie contains multiple scenes from different movies – yet this algorithm was deemed to be outside of the scope of this thesis.

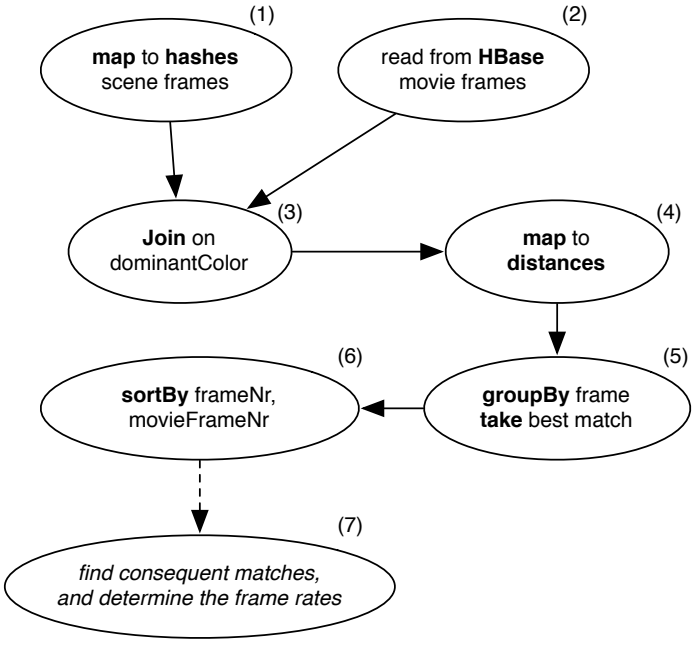


Figure 4.7: Pipeline used for Scene positioning within another movie; The last step is performed by an external application analysing the Hadoop jobs’ results.

The pipeline designed for solving this problem is shown on Figure 4.7. As can be easily observed, the steps 1 through 5 are the same (with the exception of the input from HBase in step (2) being filtered only for the targeted movie). The code used to produce these pipelines can be re-used thanks to Scala and Scalding. The 6th step though needs to be implemented a-new as it requires a new operation: sorting the matched frames in ascending order by the frame number of the analysed movie. This allows us to run an analysis tool (implemented in Scala) that will analyse the result for longest *matching sequence of frames*.

The last step (7) can be described as finding such series of frames, that all match to the same movie, and their frame numbers all increase in the same fashion as the frame numbers of the compared scene. This step was not implemented in Hadoop, and turned out to be quite challenging to be implemented resilient enough to provide easy human readable read outputs.

The problems encountered during the implementation of this pipeline highlight an interesting problem with these pipelines – extracting the actual information that is present in the computed results is often not trivial, and the noise carried with the data often complicates the tasks significantly.

4.3. Impact of big data to the design of the pipelines

The approach taken during designing the job pipelines utilises the fact of having humongous amounts of data available to match against. This also produces the problem of possible mis-matches. For example, it is completely expected and normal that some frames may match to the wrong movie. An extreme case of this being ”entirely black” or ”entirely white” frames, which appear in the vast majority of movies when fading in or out scenes, or credits. Much of the time spent on designing these pipelines was spent on trying to minimise the amount of mis-matches as well numbers of map reduce passes.

Satisfiable results can be achieved of reasonable amounts of processing time (a few hours per movie), and Hadoop’s scaling abilities help to increase computation time when tasks are able to execute concurrently.

While this section focused on the mirrored movie case, the approach developed during the system’s design scales nicely to other distortions too. Other kinds of distortions could be applied to a video, such as comparing a high-quality video with a low-quality counterpart or simple colouring filters. In fact, the tested algorithms used for hash calculation work more efficiently with blurred images – thus the distance calculation in these cases may be even more precise than in the analysed example.

Summing up, the way to design algorithms when working with big data seems to shift ones’ focus towards noise elimination as well as assuring the algorithms can be executed in parallel. Similar principles as with designing parallel algorithms apply to working with big data tasks – even theoretically expensive jobs, if parallelised, can be executed still in a timely manner.

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.

Section 5.1 describes the multitude of challenges and problems encountered while scaling Hadoop cluster. It also describes three of the applied scaling and optimisation methods in detail – including configuration changes as well as adding more nodes to the cluster.

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 overall 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 larger 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 undressed. 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 ”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.

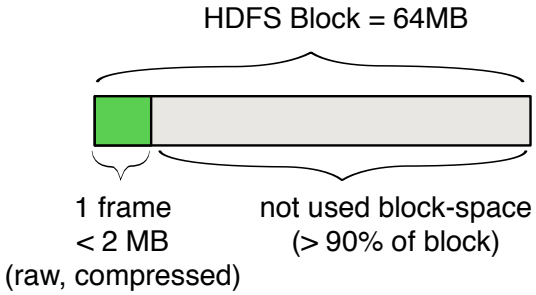


Figure 5.1: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes *name-node* to store 1 block entry in memory.

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 unnecessary 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.

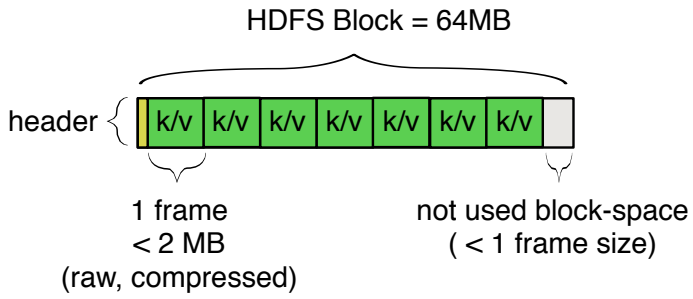


Figure 5.2: A SequenceFile allows storing of multiple small chunks of data in one HDFS Block.

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 its' 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, for increased job computation speed

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 locality* whenever possible. The concept of data locality 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, rather than the data. The rationale behind this measure is that the amounts of data are much larger than the size of application executables (jar files) so, sending the application instead of the data can save both costs and precious time.

While data locality 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 transferring the data in an ad-hoc fashion.

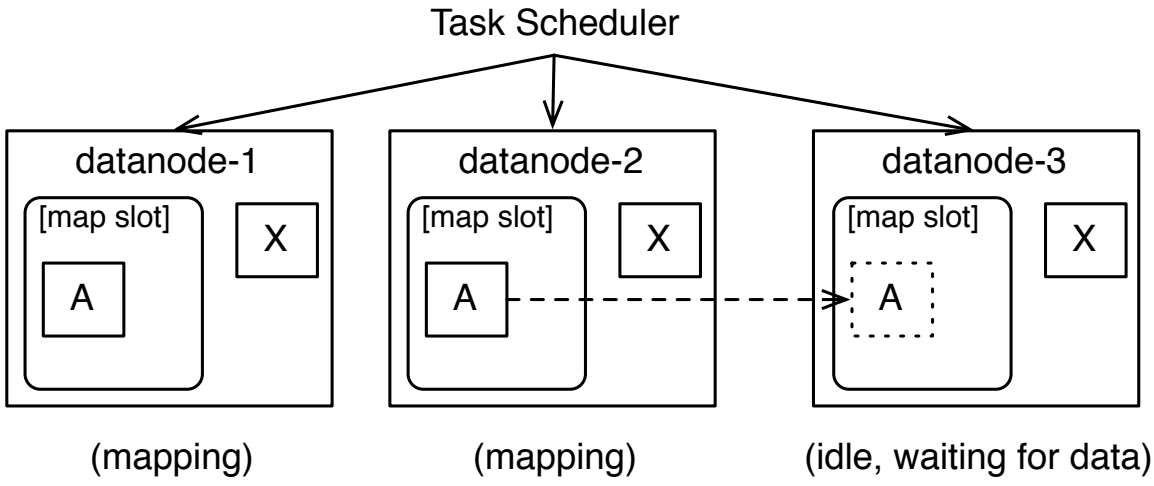


Figure 5.3: Three node cluster, with idle 3rd node; Scheduler will replicate data A while running the Job, in order to start a task requiring A on the 3rd idle node.

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 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 requiring 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 5.4) 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 experienced 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 balance the number of nodes with their fine-tuning as well as project needs. In order to test the systems behaviour, 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.

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)

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. 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 5.2. 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 average time from starting a new node on Google Compute Engine, and it joining the Hadoop cluster is *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 5.5 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 computation.

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:

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.

– **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 transferred between the two hosts, but since they are located on the same rack the transfer cost is still relatively small.

– **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 transferring 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.

Moving on to analysing the job durations of the scaled cluster, which Table 5.1 contains. The first row in this table represents the theoretical time to execute the job using only one node – the approximated 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 5.2.

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 5.1.

Analysing Table 5.1 further yields very interesting conclusions. 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

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

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

Table 5.2: Time characteristics of executing one Task of the analysed job.

previous and current configuration) is also quite interesting – especially because in cases when the replication factor was modified.

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 because of still small number 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 tasks has increased by a factor of `3.5`.

The last, and perhaps most interesting 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` percentage 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).

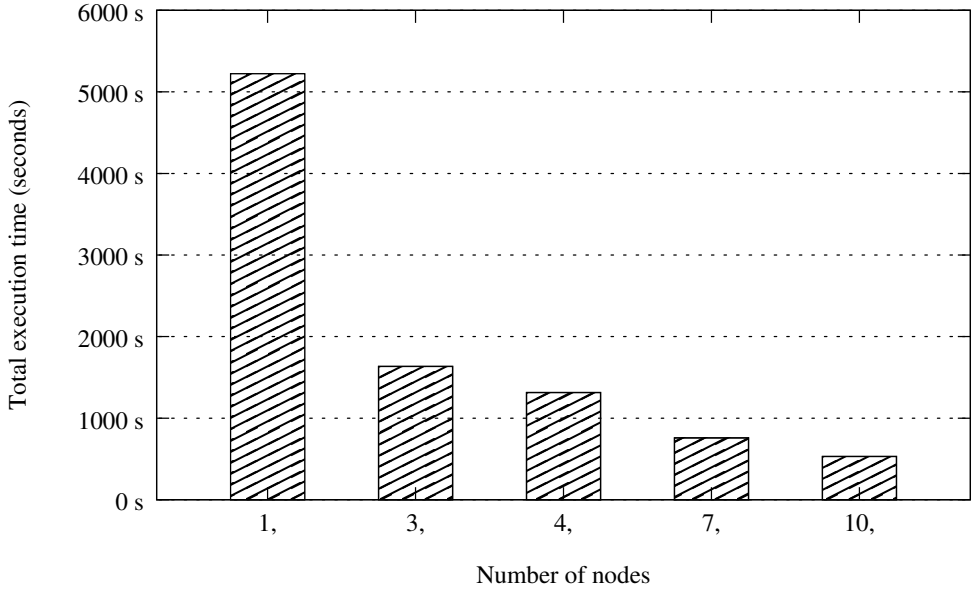


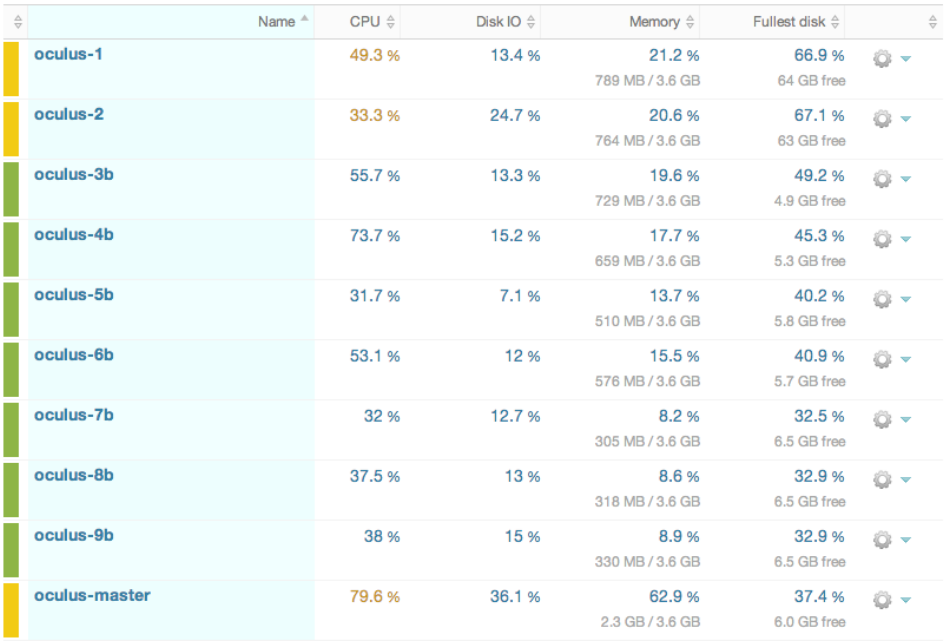
Figure 5.6: Graph displaying the total execution time of a concrete Map Reduce pipeline, in presence of *N* datanodes.

In order to summarise this section, one last observation should be made about Table 5.1, namely: in 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 linearly 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 utilisation 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 [Sam12].

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 5.8, 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



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 under-utilised 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.

scenarios as depicted on Figure 5.7, 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.

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.

In order to avoid cluster under utilisation as seen on Figure 5.7 (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 `round(1.5 * nrOfCpus)` tasks per node, yet this setting should be always customised to the workload a cluster is experiencing. 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 of 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 5.8, can be seen on Figure 5.9.

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.

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 5.7, 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 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 precise 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 challenging because of the Actor System provided by Akka was so efficient. The tasks performed by a node once it got a "download movie" message were the limiting factor, and could not been speed up doe to saturating the CPU entirely. Instead, adding new nodes to the

5.2.1. Scaling out by adding more nodes

The one optimisation 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 downloader 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.

```
1  class YoutubeCrawlActor extends Actor with OculusSelections {
2    val cluster = Cluster(context.system)
3
4    val downloadersRouter = Router(RoundRobinRoutingLogic())
5
6    override def preStart() = cluster.subscribe(self, classOf[MemberEvent])
7    override def postStop() = cluster.unsubscribe(self)
8
9    def receive = {
10     case CrawlYoutubePage(url) =>
11       val urls = extractVideoUrls(fetchPage(url))
12       urls foreach { downloadersRouter ! DownloadFromYoutube(_) }
13
14     case MemberUp(m) =>
15       downloadersRouter addRoutee downloaderSelection(m)
16
17     case MemberDown(m) =>
18       downloadersRouter removeRoutee downloaderSelection(m)
19   }
20 }
```

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). Generally speaking adding one more node would simply add one processing slot for the Download action – similarly as in the Hadoop scenario adding a node would, and since the computation is strictly CPU bound, not much can be tweaked on the system itself to scale it "up".

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 of Jobs in distributed systems – thus the conclusion is that it is easier to reason formally about a pure Actor systems behaviour, 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 [Tea14] or akka-tracing [Kho14].

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 serialising 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 remarkable 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 blindly 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

The primary goal of this thesis was to research and scale distributed media processing systems. This has been achieved by implementing a reference system, which replicated the needs and workload of a quite realistic system that would have to analyse video data. This system was then used as testbed for scaling Hadoop and Akka clusters.

The implemented reference system was able to fulfil it's initial requirements of detecting "near duplicate" movies. However the secondary requirement of finding a scenes inside of movies has been quite hard to implement and still required manual evaluation of the Map Reduce jobs' results. Several test runs using publicly available creative-commons licensed movies have been performed. The system was also tuned to perform in the face of maliciously modified video content (mirrored, blurred or otherwise distorted), and the system was still able to identify matching video data. Certain optimisations were applied to make the search times possibly fast. These optimisations required gaining knowledge about Hadoop's and HBase's internals and designing a specialised row key, as has described in Chapter 4. On the perceptual hashing front there is still much work that could have been applied to the proposed system, and better fingerprinting methods could have been applied – yes as this was not the primary focus of this thesis, the proposed system performed within it's expected expected performance boundaries of a few hours of processing per each movie.

The reference system was then used to perform multiple scalability tests and optimisations, as described in Chapter 5. The Hadoop cluster was scaled out horizontally by adding mode nodes and measuring it's performance improvement – which turned out to be nearly linear, if one trades of storage duplication for computation speed, or slightly lower if replication factors would not be tuned in accordance to the clusters size. The cluster had also undergone configuration as well as application level improvements aimed at utilising HDFS at it's best.

To conclude, the investigated distributed systems provide an astounding platform to build media analysis applications on. Akka as well as Hadoop and it's rich eco-system allow to easily build resilient and scalable applications which are able to process huge amounts of data. Hadoop especially confirmed it's ability to scale-out very easily, and has proven to be a very stable execution platform. The operational overhead of starting clusters that such applications require can be lessened by far thanks to using modern configuration provisioning utilities such as Chef, and public cloud providers make testing clusters in varying sizes very easy. In the foreseeable future we will see more kinds of these tools being perfected, as the demand for applications of such scale is still growing.

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 deployed applications on each new server in the cluster would render this process very slow and feasible. 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.1 the used cloud infrastructure 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 applications cluster Google's Compute Engine "Infrastructure as a Service" (also known under the acronym *IAAS*) was used.

Listing A.1: Creating new instance on GCE

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

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 shows the current cluster's status.

```
# 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 possible 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. An example node file is shown on Listing **??**.

Listing A.2: Example data-node.json file

```
1 {
2   "run_list": [
3     "role[base]",
4     "role[datanode]"
5   ]
6 }
```

A node description in turn relies on certain roles and recipes. Roles are defined as a list of recipes and properties that can be required together. The datanode role is fairly simple as seen on Listing **??**

Listing A.3: Example roles/datanode.rb file

```
1 run_list "recipe[hadoop::datanode]"
```

These recipes in turn refer to actual script files, which are executed ordered by their dependencies. For example, if an application's script requires a database recipe to be run, chef will reorder the run_list such that the applications' script is run after the database has been prepared. The actual execution of these scripts is shown on Listing A.4, where the process of cooking a Hadoop Datanode is presented.

Listing A.4: 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/datanode.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...
```

The use of Chef has been an tremendous help throughout the work on this thesis, and allowed to easily test various public cloud providers, without having to re-install all the infrastructure that Oculus required again.

B. Bibliography

[akk] Akka – cluster, documentation.

[ble] Blender foundation website.

[Bru05] Derek Bruening. bargraph. http://bargraphgen.googlecode.com/svn/trunk/bargraph.pl, 2005.

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

[Che13] Opscode Chef. Opscode chef. http://www.getchef.com/, 2013.

[col] Column-oriented database. http://en.wikipedia.org/wiki/Column-oriented_DBMS.

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

[con13] Concurrent inc. website. http://www.concurrentinc.com/, 2013.

[DEK77] Vaughan R. Pratt Donald E. Knuth, James H. Morris. Fast pattern matching in strings. http://www.eecs.ucf.edu/~shzhang/Combio09/kmp.pdf, 1977.

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

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

[FCG06] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Fay Chang, Jeffrey Dean and Robert E. Gruber. Bigtable: A distributed storage system for structured data. http://research.google.com/archive/bigtable-osdi06.pdf, 2006.

[ffm] ffmpeg. http://www.ffmpeg.org/.

[Fou08] Blender Foundation. Big buck bunny, movie. https://www.youtube.com/watch?v=YE7VzlLtp-4, 2008.

[Fou13] Apache Foundation. Apache zookeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index, 2013.

[Gon14] Ricardo Garcia Gonzalez. youtube-dl. http://rg3.github.io/youtube-dl/, 2006 – 2014.

[Had12] Apache Hadoop. Hadoop sequence files format documentation. http://wiki.apache.org/hadoop/SequenceFile, 2012.

[ham] Hamming distance. http://en.wikipedia.org/wiki/Hamming_distance.

[JD] Google Jeff Dean. Presentation: Taming latency variability and scaling deep learning. https://www.youtube.com/watch?v=S9twUcX1Zp0.

[JG95] Guy Steele Gilad Bracha Alex Buckley James Gosling, Bill Joy. Java programming language specification. http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 1995.

[Kho14] Lev Khomich. Akka-tracing. https://github.com/levkhomich/akka-tracing, 2014.

[mar80] Marr–hildreth algorithm. http://en.wikipedia.org/wiki/Marr%E2%80%93Hildreth_algorithm, 1980.

[MB06] Google Mike Burrows. The chubby lock service for loosely-coupled distributed systems, 2006.

[Ode14] Martin Odersky. Programming language: Scala. http://scala-lang.org, 2002 – 2014.

[PH07] Martin Odersky Phillip Haller. Actors that unify threads and events. http://lampwww.epfl.ch/~phaller/doc/haller07coord.pdf, 2007.

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

[Sam12] Eric Sammer. *Hadoop Operations*. 2012.

[sca] Twitter scalding. http://github.com/twitter/scalding.

[sec] Secondary indexes in hbase. http://hbase.apache.org/book/secondary.indexes.html. Accessed: 4.05.2014.

[SGSTL03] Howard Gobioff Sanjay Ghemawat and Google Shun-Tak Leung. The google file system. http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf, 2003.

[Tea14] Kamon Team. Kamon. http://kamon.io, 2014. Accessed 12.03.2014.

[Twi14] Twitter. Twitter.com. http://twitter.com, 2014.

[Typ13a] Typesafe. Akka – remoting, documentation. http://doc.akka.io/docs/akka/2.3.2/scala/remoting.html, 2013.

[Typ13b] Typesafe. Akka documentation. http://doc.akka.io/docs/akka/snapshot, 2013.

[Yah07] Yahoo. Hadoop at yahoo. https://developer.yahoo.com/hadoop/, 2007.

[You] Google YouTube. Youtube.com. http://youtube.com. Accessed 2013.

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