Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI STOSOWANEJ

PRACA MAGISTERSKA

KONRAD MALAWSKI

PRZETWARZANIE I ANALIZA DANYCH MULTIMEDIALNYCH W ŚRODOWISKU ROZPROSZONYM

PROMOTOR:
dr inż. Sebastian Ernst

Kraków 2014

OŚWIADCZENIE AUTORA PRACY

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

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF APPLIED COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

KONRAD MALAWSKI

PROCESSING AND ANALISYS OF MULTIMEDIA IN DISTRIBUTED SYSTEMS

SUPERVISOR:
Sebastian Ernst Ph.D

Krakow 2014

Acknowledgements...

Contents

Todo list

1. Introduction

1.1. Goals of this work

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

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

1.2. Targeted use cases

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

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

1.2.1. Near–duplicate detection

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

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

1.2.2. Data extraction

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

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

1.3. Paper structure

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

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

In the last section I will summarise the the findings.

This is not done until I'm done with the pa-per

2. Selected technologies

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

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

2.1. Hadoop DFS

As the system will require the storage of many gigabytes (hundreds) of data the core of the system will be pretty much dominated by writing and reading to / from a datastore that contains all our reference video material.

Hadoop's **??** Distributed File System was designed with such thoughts in mind, and is able to scale an abstract file system over many servers, yet providing tools to make it visible as if it was one file system.

2.2. Akka

say why akka was selected...

3. System and design overview

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

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

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

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

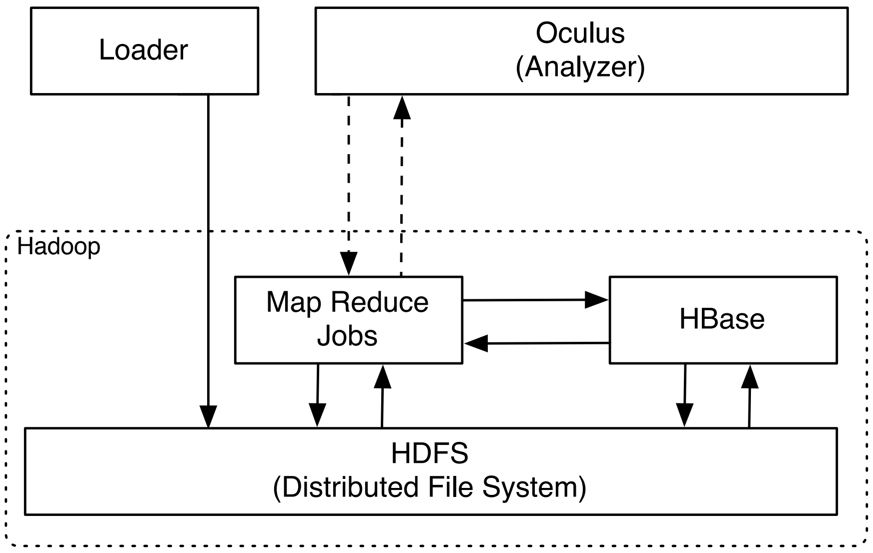

Figure 3.1: High level overview of the sys"fake tem’s architecture

3.1. Loader

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

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

The system consists of 4 types of Actors, that can react and generate only specific types of messages:

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

receives:

1 – `Crawl(siteId: String)` message

sends:

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

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

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

receives:

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

sends:

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

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

receives:

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

Since the reference data is fetched from youtube, implementing a crawler is fairly simple and requires only to fetch the site's HTML source and extract any „related" video pages (which are contained on the right hand side on the youtube website). Using this the system will automatically continue fetching similar videos, for a few *seed sites*. This is done by sending an initial message (1) via an asynchronous call to `YouTubeCrawlActor`. It will fetch the page's sources, parse out all „related video" urls and then send these to other `YouTubeCrawlActors` in the it's own local system or across the network, to another node in the cluster running another Actor System with the same types of Actors – this remoting technique has many benefits which will be discussed in the subsection 3.1.1.

Figure 3.2 illustrates what kinds of actors and messages the Loader system consists of. An important benefit for

explain in detail actor interaction

3.1.1. Spreading network IO usage by using remote Actors

This section will expand on the topic of remote actors, and what gains this has yielded to the system in general.

explain that we don't kill IO

3.1.2. Work sharing in an actor-based cluster

This sub-system is implemented the so-called „message passing style", which means that the work requests to the cluster come in as *messages*, and then are *routed* to an *actor*, who performs the work, and then responds with another message – in other words, all comunication between Actor instances is performed via messages and there is no shared state between them. This

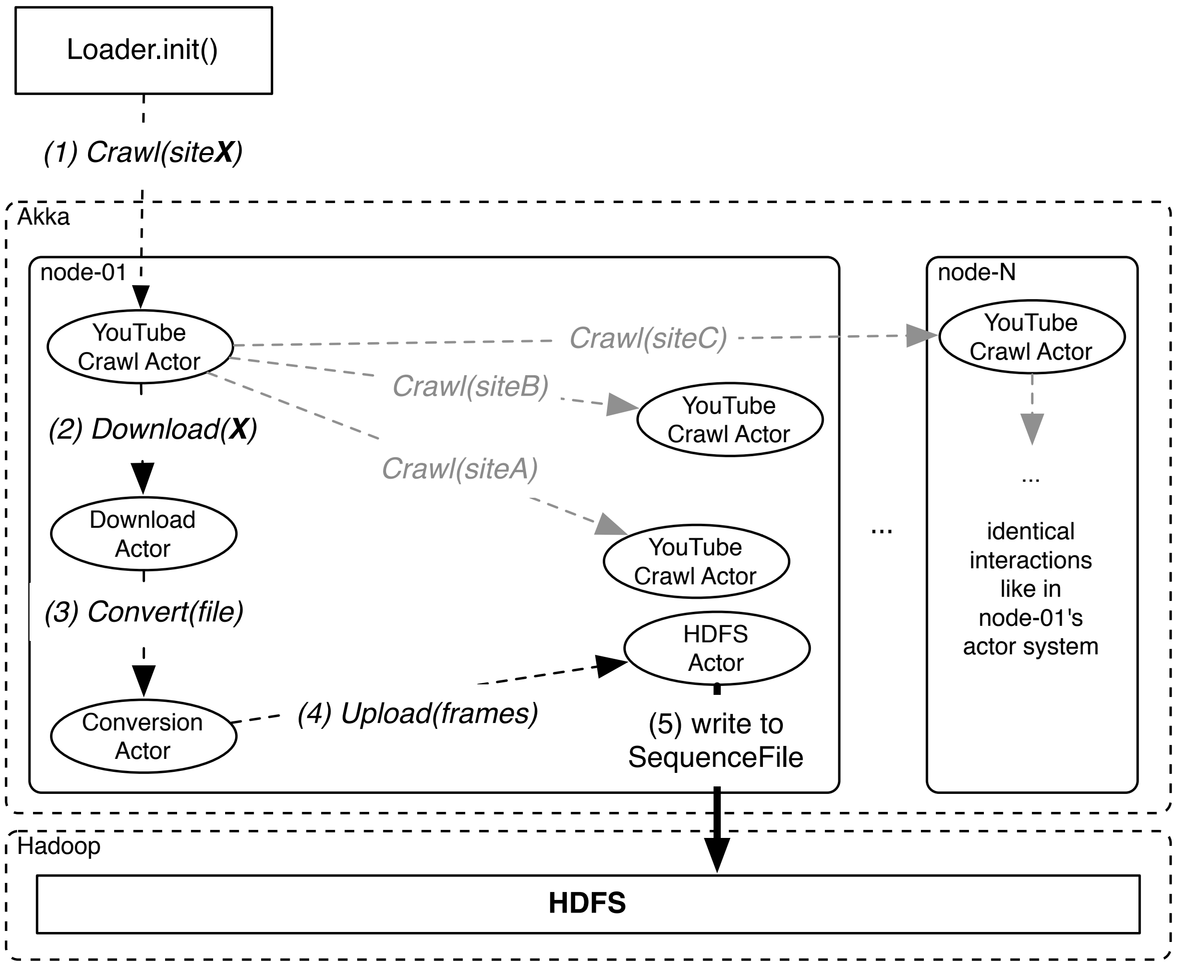

Figure 3.2: High level overview of messages passed within the Loader's actor system. Greyed out messages are also sent, but are not on the critical path leading to obtaining material from *siteX* into Oculus.

also applies to Actor instances residing in the same JVM - for the user of an „Actor System", the location (on which phisical node the actor is actually executing) of an actor remains fully transparent - this has huge gains in terms of load balancing the message execution on the cluster - the router decides who will be notified on which message, the listing 3.1 shows a typical workflow using the so-called „smallest inbox" routing strategy **??**.

Listing 3.1: smallest-inbox routing algorithm

```
                                                  ---> [inbox1, size = 3]
                                                     =>
                                                     YoutubeDownloadActor(1
YoutubeCrawlerActor -- Msg(url=http://...) --> router ---/
```

```
                                                \    [inbox2, size = 5]
                                                   =>
                                                   YoutubeDownloadActor(2)

                                /* because inbox1.size < inbox2.size */
```

The underlying Actor System is implemented by a project called Akka [**?**], and can be easiest explained as „porting Erlang concepts of the Actor Model to the JVM”. I selected the „Smallest Inbox” routing strategy instead of the other widely used „Round Robin” approach in order to guarantee not overloading any Actor with too many requests to download movies (which is a relatively slow process). Thanks to the smallest inbox routing, I can guarantee that if some of the nodes have a faster connection to the Internet, they will get more movie download requests, than nodes located on a slower network.

As mentioned before, the system is fully distributed and *any node can perform any task* submited to the cluster. For example let's take the first step in the processing pipeline in Oculus, which is `Download video from http://www.youtube.com/watch?v=-X9bcrJ3TjY` – such message will be emited by the `YoutubeCrawlerActor` and sent via a `Router` instance to an `YoutubeDownloadActor` which has the „smallest inbox”.

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.

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

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

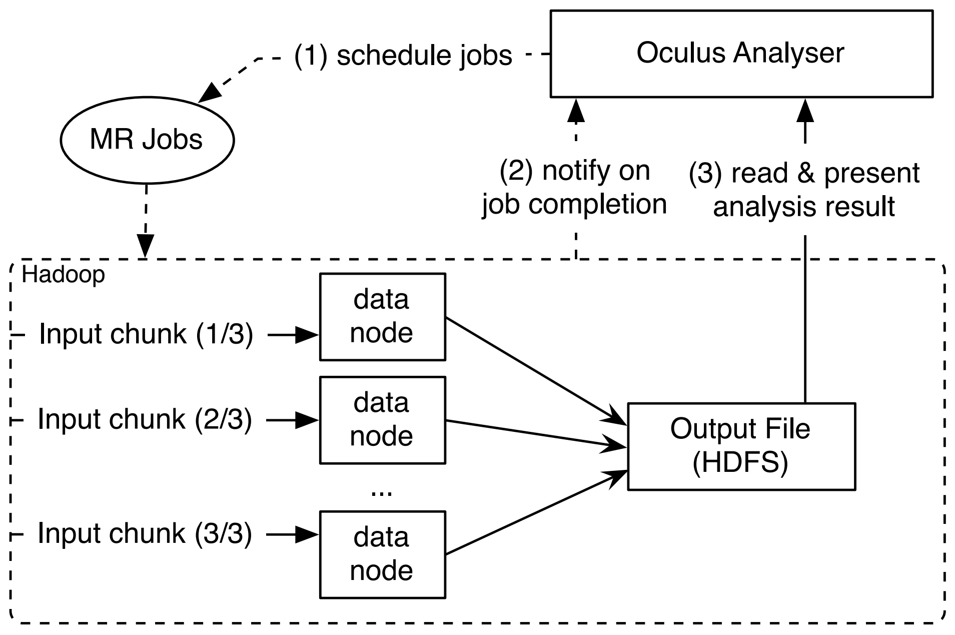

Figure 3.3: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes the *name-node* memory problems.

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

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

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

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

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

This also means that if the DFS takes a write of one file (assuming the *replication factor* equals 1) it will use up one block. By itself this is not worrisome, because other than in traditional (local) file systems such as EXT3 for example, when we store N bytes in a block on

HDFS, the the file system can still use block's unused space. Figure 3.4 shows the structure of a block storing only one frame of a movie.

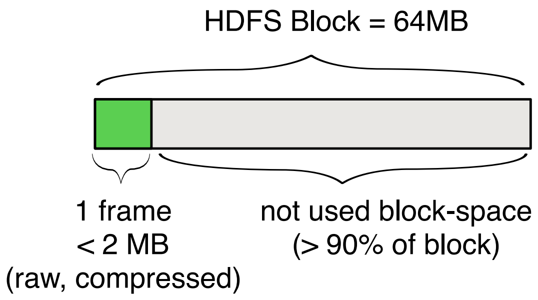

Figure 3.4: When storing a small file in HDFS, it still takes up an entire block. The grey space is not wasted on disk, but causes the *name-node* memory problems.

The problem stemming from writing small files manifests not directly by impacting the used disk space, but in increasing memory usage in the clusters so called *name-node*. The name-node is responsible for acting as a lookup table for locating the blocks in the cluster. Since name-node has to keep 150KB of metadata for each block in the cluster, creating more blocks than we actually need quickly forces the name-node to use so much memory, that it may run into long garbage collection pauses, degrading the entire cluster's performance. To put precise numbers to this – if we would be able to store 500MB of data in an optimal way, storing them on HDFS would use 8 blocks – causing the name node to use approximately 1KB of metadata. On the other hand, storing this data in chunks of 2MB (for example by storing each frame of a movie, uncompressed) would use up 250 HDFS blocks, which results in additional 36KB of memory used on the name-node, which is 4.5 times as much (28KB more) as with optimally storing the data! Since we are talking about hundreds of thousands of files, such waste causes a tremendous unneeded load on the name-node.

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

Defining Map Reduce Pipelines using Scalding and Cascading

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

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

Sequence Files

The solution applied in the implemented system to resolve the small files problem is based on a technique called „Sequence Files”, which are a manually controlled layer of abstraction on top of HDFS blocks. There are multiple Sequence file formats accepted by the common utilities that Hadoop provides [**?**] but they all are *binary header-prefixed key-value formats*, as visualised Figure 3.5.

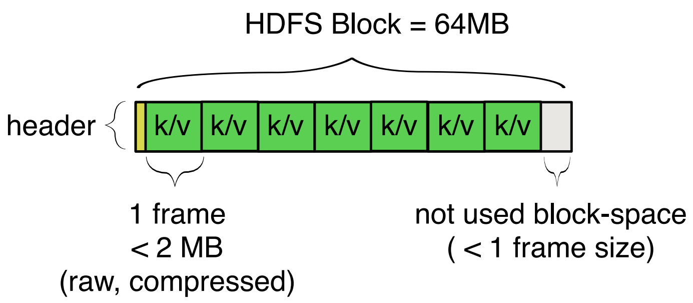

Figure 3.5: A SequenceFile allows storing of multiple small chunks of data in one HDFS Block.

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

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

4. Performance and scalability analysis

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

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

4.1. Scaling Hadoop

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

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

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

4.2. Scaling the Loader (actor system)

scale it to more nodes...

5. Results and processing times

nothing said about re- sults yet...

add some cool im- ages so it's vi- sual...

6. Conclusions

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

conclude stuff...

A. Automated cluster deployment

As part of this thesis, it

A.1. Chef - automated server configu

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

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

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

Bibliography