Fredrik Wilhelm Thon Reite

A scalable approach to video indexing and search

For use in the identification of fragmented and transformed image/video files

Master's thesis in Information Security
Supervisor: Lasse Øverlier
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Kunnskap for en bedre verden

Fredrik Wilhelm Thon Reite

A scalable approach to video indexing and search

For use in the identification of fragmented and transformed image/video files

Master's thesis in Information Security
Supervisor: Lasse Øverlier
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

Although a decade ago it was both feasible and normal to use standard relational databases for most types of data storage, the amount of generated data has since then grown exponentially, requiring new innovations to be made in the way of both indexing and processing methods. This, to enable performing video identification at a sufficient enough scale. As such we have in this thesis developed pyClipNSearchIt, which is a highly performant video identification method especially tailored for big data applications. Using a combination of smart architectural design, as well as the cutting edge FENSHES technique for performing hemming distance comparisons inside full-text NoSQL engines, we display in our approach a significant improvement over earlier available video identification methods such as PYVIDID. Specifically, in measureable areas such as speed, size reduction and accuracy. The final result is a highly scalable and cost efficient system, which can easily be used by professional and non-professional actors alike.

Sammendrag

Selv om det for et par år siden var både mulig og normalt å bruke tradisjonelle relasjonsdatabaser for de fleste typer datalagring, har det i de senere årene blitt vanskeligere og vanskeligere å håndtere eksponentielt økende datamengder. Det har derfor måttet blitt gjort nye innovasjoner innenfor video identifikasjons feltet, for å muliggjøre både rask indeksering og behandling av data i større skala. En av disse innovasjonene er pyClipNSearchIt, som er en ny video identifikasjonsmetode skreddersydd for bruk i prosesseringen av "big data". Dette har vi oppnådd gjennom en kombinasjon av smart design, samt bruk av metoden FENSHES for å gjennomføre hemming distanse kalkulasjoner direkte i NoSQL søk. Konsekvent har dette medført en betydelig forbedring innen målbare områder som hastighet, størrelsesreduksjon og nøyaktighet i sammenligning med tidligere kjente metoder som PYVIDID. Det endelige slutt resultatet er derfor et mer skalerbart og kostnadseffektivt video identifikasjons system, som enkelt kan benyttes av både profesjonelle og ikke-profesjonelle aktører.

Acknowledgements

I would first and foremost, like to thank my supervisor Lasse Øverlier for his many thoughtful comments and feedback throughout the entire semester. This thesis would not be possible without his help. I would also like to state my sincere appreciation of Matylda Stefanska for taking the time out of her busy schedule to review and correct my many, many rough drafts. Lastly, I extend a heartfelt thanks to my Dad for his as always - unwavering support. Thank you all.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

1.1 Covered Topics

In database systems the term "indexing" refers to the process by which data is stored to improve data query and retrieval speed. The most common method by which this is done is key-value association. Both traditional SQL databases and new "NoSQL" databases support this type of indexing, but the latter is typically more suited for the storage and processing of big data. This is because NoSQL databases scale better horizontally at the cost of consistency; also known as the concept of eventual consistency.

In this thesis, we wish to take advantage of the various benefits that a NoSQL database brings over a traditional SQL database, presenting a new approach to improving the scalability and speed of video/image content identification over previously published research [1]. The topics covered by this project will therefore range from perceptual hashing to data storage and size reduction of big data.

1.2 IEEE Keywords

Image recognition, Image databases, Multimedia databases, Scalability, Indexing, Big data applications

1.3 Problem Description

Whilst a decade ago it was both feasible and normal to use a standard relational database for most types of non-blob data storage, the volume of data has since then grown exponentially [2], requiring new innovations to be made to meet the storage and processing requirements of today. As such, the question of "Does it scale?", i.e. scalability, is no longer a strange or unfamiliar concept, but rather something all "Big Data" companies have to deal with on a daily basis. In particular, scalability has been essential in the area of content indexing and identification, something especially Youtube has experienced with its growth as a platform and the development of the Content ID system [3]. However as it currently stands, there are few or no easily accessible methods for creating systems of a similar size and purpose that other companies and the open source community might take advantage of.

1.4 Justification, Motivation and Benefits

One of the primary motivations for attempting to solve the problems listed in the previous section, is to allow more sectors to make use of image/video content identification systems. Existing products are either proprietary, struggle with a large number of data, or only target a niche audience. This in turn, makes it harder for companies to tailor a potential solution to their specific use-case without expending a large amount of resources. To remedy this, an open source solution that easily scales in accordance with a company's data size is a key candidate for a master's thesis.

1.5 Research Questions

In conjunction with the project supervisor the following research questions have been defined as relevant to solve the research problem outlined in section 1.3:

- R.1 Which effective techniques exist for performing content identification?
- R.2 How do existing solutions tackle the problem of big data in terms of scalable content identification and storage?
- R.3 How can the overall size of indexed datasets be optimized?
- R.4 What can be done to improve the hashing, indexing and search times for very large datasets?

In essence the focus is on determining how *others* have tried to solve the problem, and how *this* project will differ in its solution. Therefore, the answers to the latter two questions are key to the value and quality of the resulting thesis.

1.6 Planned Contributions

The intended end result of this master's thesis is a better and more accessible method of creating an easy-to-use scalable video/image indexing database for use in the search and identification of fragmented and transformed files. Although there already exist some frameworks and methodologies aiming to achieve the very same goals, e.g. Griffeye [4] and PhotoDNA [5], this thesis will differ in its open source approach based on new research [6], allowing a broader community to make use of the final results.

Chapter 2

Background

In this chapter we discuss the background and related work underlying our thesis' presented results, exhibiting how previously published papers and technical solutions can come together to serve as the starting point for what is new and exciting experimental research; as is the case with our approach to a scalable video indexing and search method for use in the identification of fragmented and/or transformed image/video files. In summary, this involved looking into what the already established methods for performing content identification were, and then researching how these methods were respectively used in different video/image identification systems as GriffEye, Youtube Content ID, PYVIDID, etc. With regards to the paper's topic of scalability, we also investigated how each of these systems were built to store and process - both successfully and unsuccessfully, large amounts of data.

2.1 Content Identification

For ease of understanding, all of the available content identification methods that we explore in this paper are divided into one of the following two categories: Traditional methods and Machine Learning (ML) based methods. The first category classifies methods that do not contain a prerequisite "learning" step, instead identifying image/video files purely based on simple[1] algorithms which extract and then compare individual file features against each other. Machine learning based methods on the other hand, **will** require the initial model to be trained on a prior dataset, but can arguably give more accurate results by operating as black boxes that retain knowledge over time. That's not say that one approach is better than the other - as speed, simplicity and scalability are also important factors when contemplating a traditional versus machine learning based content identification approach. Notably, the latter may also reuse or combine elements from the former to establish the features that are used to train the model.

[1]Here we are referring to the fact that they do not retain knowledge from run to run

2.1.1 Traditional

Watermark

One of the earliest known approaches to performing content identification is the visual watermarking method that was first developed in Fabriano, Italy, in 1282 [7]. At the time of its invention, the process involved raising the thickness of printed paper whilst it was still drenched in water to create an opposing effect of visible lighter and darker spots, making it easy to discern legitimate from illegitimate prints. As time passed however, both the original watermark process and term, evolved to encompass more than simple thickness variability. With the *Dandy Roll* process invented by John Marshall in 1826 it became possible to impress water-coated metal stamps onto paper, and the *Cylinder Mould* process developed in 1848 allowed users to create high precision greyscale prints of varying density. The most recent innovation in the world of watermarking is digital watermarking; the term coined by Andrew Tirkel et al. in the 1992 paper "Electronic Water Mark" [8].

In contrast to traditional physically based watermarks, an electronic/digital watermark can consist of more than just visual artefacts. For example, in one case a digital watermark can be a simple binary blob that is hidden deep inside a file, whilst in another it may appear as a semi-transparent logo in the corner of a digital image or video frame. Most commonly, this type of watermark is used for the purpose of enforcing material copyright protection, but there are also many other examples of relevant use-cases such as military document classification and forensics identification. In any case, digital watermarks can serve as a perfectly valid option for content identification, even retaining the ability to identify fragmented clips from longer videos if the watermark is visual in nature. That being said, the latter approach is definitely harder to detect algorithmically and at scale without the help of machine learning. Additionally, it should be noted that visual watermarks have been criticized by many researchers as being not robust and easy to remove [9][10][11].

Signature

Another popular method for identifying digital content, is to transform a given input's individual and distinctive features into one or more (relatively speaking), compressed signatures. For instance, by taking the binary content of a file and then running it through a hashing function like SHA256, it is possible to generate an unique 256 bit long signature that will always represent the contents of **that** particular file. However, as a potential following downside, the cryptographic nature of SHA256 will result in any bit size changes to the original input creating an entirely different output signature. This unfortunately makes the described type of hashing function nigh unusable for comparing the contents of two similar but not identical files. Therefore, as a work around to this problem, one instead has to use a *fuzzy* type of hashing function.

Fuzzy hashing is a category of non-cryptographic algorithms that aims to produce similar hashes/signatures for inherently closely related inputs. In practice, this means that the algorithms do not have a cryptographic diffusion property to them, but may still to a small degree keep the ability to hide the relationship between an originating entity and a produced hash. Commonly, the technique has been used in everything ranging from malware classification and information clustering to copyright enforcement and image similarity comparisons [12][13][14]. The latter use-case is often based around hemming distance comparisons[2] and operates on a sub-category of fuzzy hashes, namely perceptual hashes.

Perhaps the most widely known implementations of perceptual hashing are *aHash* and *pHash*. Both of these methods run on a per. image/frame basis to compress the displayed visual content into a corresponding hash signature. aHash's approach is based around averaging the high and low frequencies of an image by reducing its size and then converting the remaining pixels to grayscale. Afterwards, a 64 bit hash can then be calculated by comparing the color value of each pixel in the now 8x8 image against the mean color of the image. Pixels with a color value above the mean will have their bit set to 1, whilst those below will be 0. The act of generating a pHash is unfortunately a bit more complicated, but can in return offer more much more robust and accurate results than aHash [15]. First, the image is resized to 32x32 pixels before a Discrete Cosine Transform (DCT) is applied to convert the image into a spectrum of frequencies. Only the top left 8x8 of the resulting scalars are then kept to construct the final 64 bit hash. This is done the exact same way aHash does it, just this time substituting the pixel colors for DCT frequency values.

With regards to scalability, hemming distance comparisons between perceptual hash signatures have traditionally been done on a single machine. Although this is fine for small datasets, if the goal is to scale up a content identification system to identify images inside a database consisting of potentially thousands or **millions** of videos, processing and memory resources will quickly turn scarce. As such, in 2019 some researchers decided to look into options for performing r-neighbour searches in hamming space, directly inside of the backing database cluster. This method is known as FENSHES, and takes advantage of the capabilities offered by the NoSQL database ElasticSearch to process large datasets very quickly and return the lowest scoring matches. [6].

[2]The hemming distance between two binaries of equal length is the number of bits/positions at which they differ, `https://en.wikipedia.org/w/index.php?title=Hamming_distance&oldid=1079111331`

2.1.2 Machine Learning

Classic

A classic machine learning approach to performing content identification involves preprocessing some input by means of e.g. principal component analysis[3] (PCA), and then feeding the output through either a state vector machine (SVM), neural network (NN), or other similar types of ml models to interpret the given data. Typically this interpretation may come in the shape of a fingerprint/signature [16], a label classification [17] or a cluster [18], but other types of outputs can also be expected depending on how the used model is implemented. The only real requirement is that the result should in and of itself represent an *understanding* of the input's underlying features.

In the paper "A Comparative Study of Support Vector Machine and Neural Networks for File Type Identification using n-gram analysis" from March 2021, Joachim Sester et al. showcase in their experiments that although most classic machine learning models can provide a high degree of accuracy when used for content identification, many of them will struggle with regards to scalability [19]. As such, the authors ended up giving no clear preference to any of the tested ml models in their final results. Instead, they concluded that deep learning was the best candidate for any future and better content identification approaches; specifically, file type content identification methods.

Deep Learning

Deep learning is a modern[4] subset of the broader machine learning family, characterizing multi-layered artificial neural networks (DNN) that consist of three or more processing layers [20]. It is an attempt at replicating how biological systems typically process and interpret information; spitting out predictions based on what the learned most valuable data features are. In the human brain, this process of learning is represented by the creation and interaction of neurons, whilst it in the DNN is represented by the fitting of the model's multiple layers (neurons) through backpropagation and/or gradient descent[5]. This fitting is either done supervised or unsupervised. The former approach operates on pre-labeled datasets, whilst the latter operates on unlabelled datasets - discovering hidden patterns in the processed data without requiring human intervention.

[3]This is commonly done as a way of reducing data dimensionality by projecting multiple data points onto just a few principal components, `https://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=1088263743`

[4]Relatively speaking with regards to the current machine learning trends anno 2022, `https://www.marktechpost.com/2022/01/13/top-deep-learning-trends`

[5]Gradient descent is an optimization algorithm for finding the minima(s) of a differentiable function, `https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=1087292976`

With this inherent capability for understanding data, along with recent advances made in GPU AI-enhanced computing [21], deep neural networks have become a staple in many of the content identification systems used around the world. China for example, has adopted a DNN model for recognizing the faces of its citizens [22][23], and Facebook employs a similar method to automatically tag friends in uploaded pictures [24]. This goes to prove to a certain extent, the current usability and scalability of DNN based content identification methods, although with some associated ethical concerns [25].

2.2 Existing Solutions

To date, a wide variety of different solutions have been proposed to allow users to identify fragmented and/or transformed images/videos from large datasets. Many have already been employed to great success within fields such as digital forensics and copyright enforcement [4][26], whilst others have remained relatively unknown and unused, mostly consisting of purely theoretical approaches. The primary differentiating factor however, remains the individual solution's ability to be utilised at *scale* with a high degree of *accuracy*. As such, one of the questions we wanted to answer in our thesis was how existing solutions approached these problems and attempted to solve them.

2.2.1 Griffeye

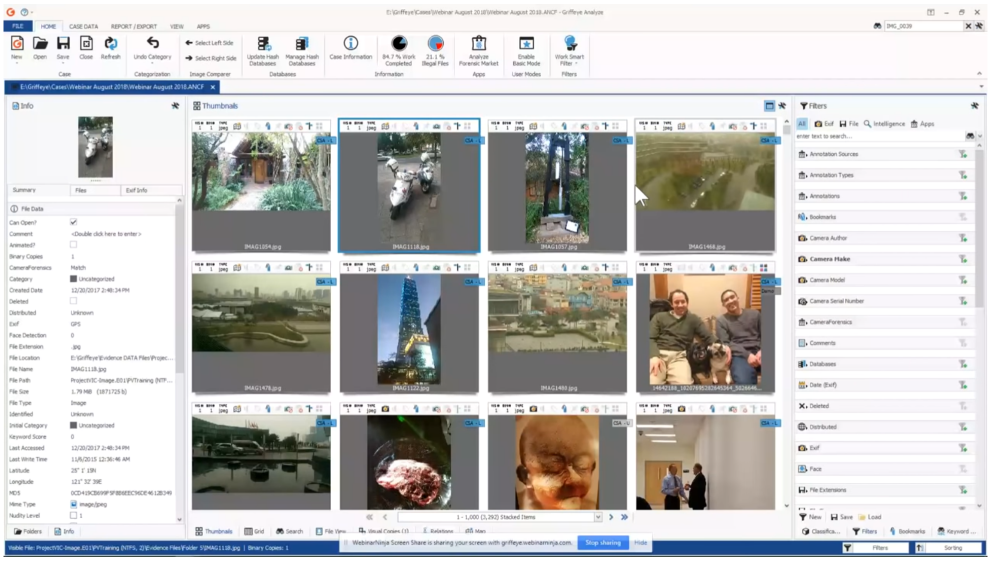
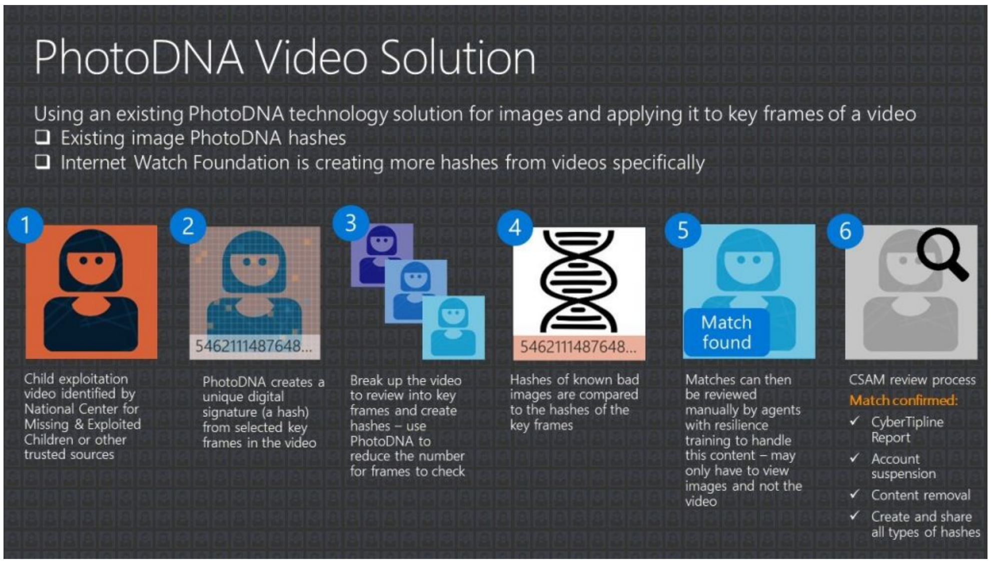


Figure 2.1: The user interface of GriffEye [27]

GriffEye Analyze is a professional video/image identification tool used by numerous law enforcement agencies all around the world to aid with criminal forensic investigations [4]. It was created to enable the processing, sorting and analysis of large collections of data, supporting content indexing and identification through features such as image content labeling, facial recognition and automatic classification of child sexual abuse [28]. Supporting the endeavour, is the Griffeye Processing Engine, which consists of an AI component (GriffEye Brain) operating on an interconnected network of databases named the "GriffEye Intelligence Database" (GID). This network allows multiple different national and international agencies to collaborate on cases by simply opting to share their individual databases with each other, letting GriffEye take care of the necessary audit log protocols and data boundaries [29].

To conform to the specific use-case, GriffEye Analyze is split into three different suites of product: Analyze CS Enterprise, Analyze CS Operations and Analyze DI Pro. Although named differently, all of them share the same underlying GPI engine, GIDs, and graphical interface (see figure 2.1) which the end-user will make use of. The only difference exists in the number of simultaneous users and cases that each of the products support. Unfortunately, none of them are open source making it hard or impossible to measure their performance in terms of scalability and accuracy outside of black-box testing.

2.2.2 PhotoDNA


PhotoDNA Video Solution

Using an existing PhotoDNA technology solution for images and applying it to key frames of a video
- ❑ Existing image PhotoDNA hashes
- ❑ Internet Watch Foundation is creating more hashes from videos specifically

1. Child exploitation video identified by National Center for Missing & Exploited Children or other trusted sources
2. PhotoDNA creates a unique digital signature (a hash) from selected key frames in the video
3. Break up the video to review into key frames and create hashes – use PhotoDNA to reduce the number for frames to check
4. Hashes of known bad images are compared to the hashes of the key frames
5. Matches can then be reviewed manually by agents with resilience training to handle this content – may only have to view images and not the video
6. CSAM review process
   Match confirmed:
   - ✓ CyberTipline Report
   - ✓ Account suspension
   - ✓ Content removal
   - ✓ Create and share all types of hashes

Figure 2.2: The official explanation of how PhotoDNA works according to a Microsoft blog article [30]

PhotoDNA, shown in figure 2.2, is a technology marketed by Microsoft for finding and removing images and videos known to be associated with cases of child exploitation and pornography [5]. It was originally developed in partnership with

Dartmouth College in 2009 as a new (claimed) non-reversible type of perceptual hashing algorithm, creating unique photo signatures that digital investigators could use to identify potential abuse victims [31]. Later the solution went on to be donated to the National Center for Missing Exploited Children (NCMEC) in the United States to allow other companies, such as Facebook and Apple, to also integrate the system into their services [32][33]. However, supporting PhotoDNA at such a large scale has not been easy, and Microsoft therefore only serves it as an Azure Cloud Service [34]. To use the service, organizations must first be vetted and qualified by a third-part vetting service. Then, afterwards, the organization may upload images to the cloud to hash them and find out if a match exists in the central NCMEC database.

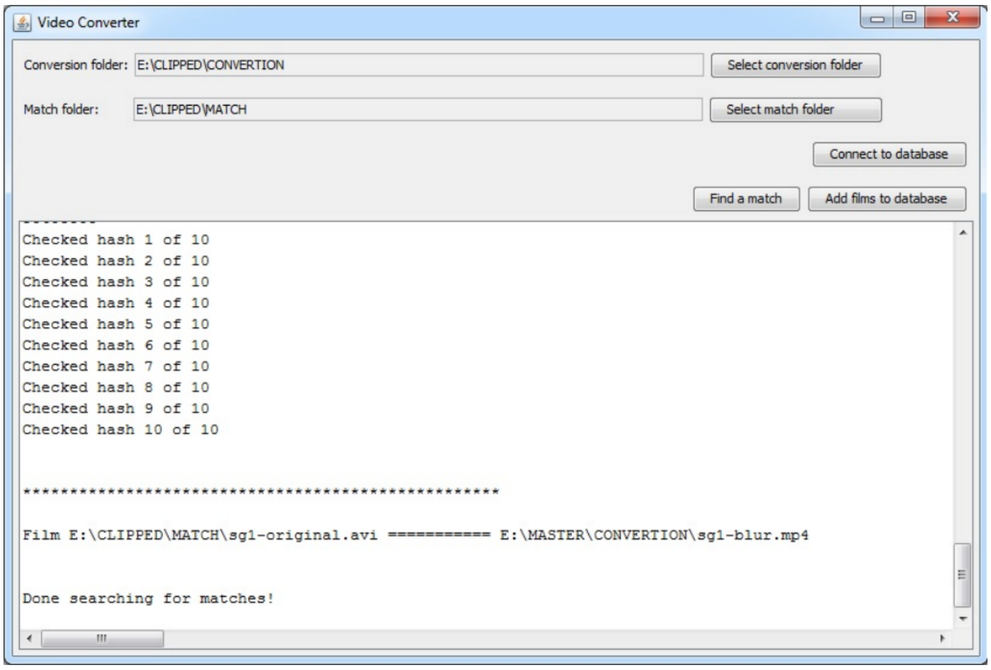
Although the actual perceptual hashing algorithm that PhotoDNA uses has never been disclosed to the public, a man by the name of Jan Kaiser managed in August 2021 to recreate it through trial and error based on a telling blog article published earlier that same August by the researcher Neal Krawetz [35][36]. In his article, Krawetz revealed that whilst the PhotoDNA algorithm is claimed to accurately generate similar hashes for visually similar inputs, changes to less than 2% of any of them will void any detection comparisons. Another flaw in the algorithm is its claimed "non-reversiblity" which has been proven factually untrue [37].

2.2.3 Youtube Content ID

Content-ID is Youtube's copyright enforcement system, providing a closed-door approach to content identification and management for various different types of copyright owners. Using it, they are able to upload any digital media which they own the rights to, and then tweak how the system will respond to videos uploaded by users containing the aforementioned copyrighted content [38]; most often without the original owner's permission. Based on Youtube's own description, this works on the basis of some type of audio and/or video fingerprinting method, presumably backed by a deep machine learning AI that is responsible for processing the vast amounts of user derived content on the website [26]. Though, this last point primarily boils down to pure speculation with no official sources disclosing in detail how the system actually works - not even to the copyright owners themselves.

What is known however, is that Content-ID has proven to be highly scalable - processing and matching against an estimated 500 hours of uploaded content each minute [3]. How accurately it identifies content though, is another question - with Joanne E Gray et al. as well as many partnered content creators voicing their concerns relating to the removal rates of what can be considered fair and transformative uses of identified snippets of content [39]. In the paper "Adversarial Attacks on Copyright Detection Systems", authors Parsa Saadatpanah et al. also regarded the system as vulnerable to possible perturbations with non-adversarial random noise in audio tracks [40].

2.2.4 CLIPPED



```
Checked hash 1 of 10
Checked hash 2 of 10
Checked hash 3 of 10
Checked hash 4 of 10
Checked hash 5 of 10
Checked hash 6 of 10
Checked hash 7 of 10
Checked hash 8 of 10
Checked hash 9 of 10
Checked hash 10 of 10


****************************************************

Film E:\CLIPPED\MATCH\sg1-original.avi =========== E:\MASTER\CONVERTION\sg1-blur.mp4


Done searching for matches!
```

Figure 2.3: The user interface of CLIPPED [1]

CLIPPED was a video identification program developed and published by Kjetil Gardåsen in 2013 to submit and match short video clips against likely root file sources [41]. It accomplished this task by comparing a given video input against a number of indexed video frames which had previously been extracted, hashed and stored in a HyperSQL[6] database. The most likely video origin would then be determined by picking the matched frame hashes with the lowest scoring hemming distance between them. For the actual frame processing, the program used FFmpeg[7] to extract images at a rate of 1 frame per. second of video footage, and then hashed the resulting output using pHash to produce 49 bit long frame signatures.

Reportedly, CLIPPED managed to achieve a 100% true positive rate for the few fragmented clips which Gardåsen matched against a 140 hour dataset in his original experiments, but these results should be taken with a grain of salt

[6]A Java SQL relational database, `https://en.wikipedia.org/w/index.php?title=HSQLDB&oldid=1081719211`

[7]A highly portable framework for decoding, encoding, transcoding, etc. different visual and audio medias, `https://ffmpeg.org/about.html`

considering the project is not available on Github anymore for reproducible testing. Instead, the project has become entirely superseded by its spiritual successor, PYVIDID.

2.2.5 PYVIDID

PYVIDID can be seen as a natural continuation of the previously presented CLIPPED video identification project, and was created by Ola Kjelsrud to address some of the issues present in the original implementation of the program [1]; with Gardåsen's blessing of course. In particular, Kjelsrud wanted to fix the multithreading issues causing CLIPPED to go into a deadlock upon hashing videos, and attempt to improve the perceived general speed of the program. More extensive testing was also performed to verify the suggested method's robustness, accuracy, speed and temporal localization ability, showcasing improvements in nearly all areas over previous approaches.

In contrast to CLIPPED's Java codebase, PYVIDID is primarily written in the Python programming language with an included specialized Java component solely reponsible for speeding up video searches. As a new feature, Kjelsrud added support for temporal localization of fragemented clips in longer videos with incorporated timestamps, and made it easier to "swap" the used perceptual hashing algorithm pHash for other similar functions such as aHash and dHash. However, the program unfortunately does not carry over CLIPPED's original graphical user interface as shown in figure 2.3.

In terms of scalability, Kjelsrud's program is not really an improvement over CLIPPED. Both programs use a traditional SQL database to index processed videos[8], and both have to load the frame table into memory to perform the necessary hemming distance similarity calculations. Therefore, at a very large scale, the proposed solution will either slow down as a result of the number of SQL select statements increasing, or the fact that hash comparisons may not easily be processed in parallel.

[8]Sqlite in PYVIDID's case

Chapter 3

Methodology

The overarching purpose of this chapter is to provide a certain amount of credibility to the thesis by outlining a set of easy-to-reproduce requirements and steps by which all of the results shown later in Chapter 4 can be tested. This ensures that other researchers can utilize the same research methodology, setup, configuration and dataset as the author in writing to verify any of the presented experimental results. It also might be used to explain somewhat minor variations that can originate from user-differences in crucial lab environment components such as:

- Frontend hardware
- Backend hardware
- Configurations

Below, the chapter has been divided into four different sections, each detailing a separate aspect and part of the thesis' workflow. First, the general research approach and philosophy is explained, specifically regarding how the related work was found and later dissected and then utilized. Secondly, the setup specifications and configuration for developing and testing the developed program is discussed, showing how seemingly small changes to initial parameter values might impact the overall speed and accuracy of the program. Lastly, a short description is given of the downloaded dataset, as well as how it was processed to produce quantifiable results for comparison with earlier available methods of fragmented video and image identification, such as PYVIDID and CLIPPED.

3.1 Research

The core philosophy, or rather, methodology behind this thesis can best be described and characterized as one of discovery, experimentation and verification. More widely known, it is also referred to as the *general scientific method*, which by the Oxford Dictionary is defined as "*The approach that science uses to gain knowledge, based on making observations, formulating laws and theories, and testing theories or hypotheses by experimentation*" [42]. This means that by following the method, it is possible to ensure that any given experimental results remain reproducible and verifiable even in independent lab environments, thus increasing their overall reliability.

We initially started our project's research phase by performing a thorough review of existing literature that could help us with understanding the outlined research questions and problems. This entailed *collecting* useful information from various different sources such as research papers, books and online websites. All of the collected data was then *processed* to determine any valuable artefacts that might help serve as a baseline for the actual content of the thesis. Finally, the utilized sources were presented in section 2 to substantiate the paper's background and give credit where credit is due.

3.1.1 Collection

To source knowledge of a high enough standard as well as ensure a trustworthy origin, a combination of the on-campus library, Google, Oria[1] and Google Scholar were used to search for topics related to our IEEE keywords. The latter two search engines allowed access to a wide variety of academic papers and articles from all over the world, often with included notations for peer-reviews, citations and references, which proved especially useful for later processing.

3.1.2 Processing

Verifying the collected data was mostly done using a qualitative approach, where any findings or results were reproduced or tested locally to confirm their reliability and validity. This helped differentiate real claims from fake ones, and reduced any potential time spent chasing unrealistic goalposts.

3.1.3 Presentation

For the purpose of presenting the related work, relevancy was the factor that was emphasized the most. This meant that some information had to be left out to better highlight the major key points in the processed data. In particular, special focus was put on displaying essential and relevant information instead of writing simple high-level overviews/abstracts.

3.2 Setup

Although the project's lab environment is very much an intertwined setup, it can, for the sake of explanation be separated into what is essentially two primary components: a frontend component and a backend component. The former encom-

[1]A search engine used by many Norwegian universities to provide access to various different research materials, `https://bibsys-almaprimo.hosted.exlibrisgroup.com/primo-explore/search?vid=BIBSYS`

passes all of the tools and devices that were used to develop and employ the proposed method and Python program, whilst the latter covers the interfaced database that everything else was built around. For both components, NTNU Gjøvik's Openstack[2] platform "SkyHiGh" was heavily made use of to satisfy the horizontal scaling requirements of NoSql databases.

3.2.1 Frontend

The frontend setup consists of two different machines: an on-premise Windows desktop and an off-premise Ubuntu node. For the development of the application, the Windows platform was chosen due its large ecosystem of integrated development environments (IDE), as well as a general sense of familiarity with working on the platform. General testing on the other hand, took place on the virtual Ubuntu linux node to make inter-node communication within the Openstack network easier, in addition to lacking the dedicated CPU/GPU resources required to speed up the hashing phase of the program. Note however, that the last part is both possible and *recommended* to amend in a production deployment, thus decreasing the number of frontend machines to just one; preferably only a Linux node.

To set up the necessary software environment on a client, a simple Python script named *setup_frontend.py* is provided. Depending on the operation system, of which Windows and Debian based Linux distributions are supported, the script will find, download and install all of the pip, apt and FFmpeg packages needed to run the main program. A default configuration file is also created called *config.json* to allow the user to tweak behavioural parameters of the program. The entirety of the setup script can be viewed in appendix A.

Specifications

Below are two tables, 3.1 and 3.2, containing the exact specifications of the previously mentioned Windows and Linux clients. Notably, only the Python interpreter version >3.9.1 is considered a **hard** requirement[3], with the other requirements being marked as optional or possible to substitute.

[2]An open source software for cloud infrastructure deployment, `https://www.openstack.org/`

[3]This is due to the use of the new dict merge functionality in 3.9, `https://docs.python.org/3/whatsnew/3.9.html`

Table 3.1: Desktop Windows specifications

Hardware	**Name**
CPU	Intel i5-9600K
GPU	EVGA GeForce GTX 980 Ti SC GAMING ACX 2.0+
Ram	Corsair Vengeance LPX DDR4 3000MHz 16GB
Storage	WD Green SSD WDS480G2G0A
Storage	ST1000DM003-9YN162
Software	**Name**
OS	Windows 10 Pro 19043.1645
IDE	Visual Studio Code 1.62.3
Interpreter	Python 3.9.2
Media Framework	FFmpeg 5.0

Table 3.2: Virtual Ubuntu master node specifications

Hardware	**Name**
CPU	VCPU 2 Core @ 2099.998 MHz
GPU	VGPU (Cirrus Logic GD 5446)
Ram	VRAM 8 GB
Storage	SSD 40GB
Software	**Name**
OS	Ubuntu 18.04.6 LTS
Interpreter	Python 3.9.12

Configuration

Code listing 3.1: Frontend component of the pyClipNSearchIt configuration file

```
{
    "ffmpeg": {
        "use_hwaccel": false,
        "capture_every_n_seconds": 1
    },
    "search": {
        "max_size_response": 15,
        "r": 5
    }
}
```

In the context of configuration, there are 4 different settings in *config.json* that affect or alter the behaviour of the frontend interface. Listing 3.1 displays which ones these specifically are, with a respective explanation for each one given in the list that follows:

1. **use_hwaccel:** Use Nvidia GPU hardware acceleration. Defaults to false because CPU processing is usually more accurate and faster.
2. **capture_every_n_seconds:** Capture an image frame every *n*th second of the video. Defaults to 3 seconds to limit indexed document size, but setting it to 1 or 2 can potentially make hashing and searching more accurate.
3. **max_size_response:** Determines the max number of hits returned by a single image search. For videos, this limit applies on an individual frame basis, with duplicate hits later being removed. Defaults to 15, but decreasing it more can increase processing speed at the cost of accuracy.
4. **r:** The hemming distance threshold at which search hits are discarded. Defaults to 5, but the value can be tuned higher to accommodate comparisons between very different images at the cost of potential accuracy.

3.2.2 Backend

Contrary to the frontend portion of the lab environment, the backend setup is entirely virtual and is made up of 3 hardware-identical Ubuntu nodes. Together, these nodes combine to form a small Elasticsearch cluster consisting of one master/data node and two data/ingest nodes. We opted to use Elasticsearch, largely on the basis of its ingest pipeline features, scriptable queries, prevalence in academic papers and portability support[4]; but previous experiences with using the official Javascript and Python client APIs also played an important role during the consideration process.

To set up the requisite Elasticsearch indexes, pipelines and scripts, simply run the *setup_backend.py* script attached in appendix C. Appending an additional –*force* argument to the command line will result in any existing data being discarded before the setup process starts.

Specifications

Table 3.3 shows the hardware and software specifications of the virtual Ubuntu machines that make up the NoSQL Elasticsearch cluster. These specifications are based off Elastic's official recommendations for a small database deployment [43] [44], but can be tweaked further to better accommodate the individual user's needs and resources. The only **hard** requirement is that a cluster must contain at least one or more designated ingest nodes.

[4]Here we are referring to the index migration and sharing possibilities, `https://www.elastic.co/guide/en/cloud/current/ec-migrating-data.html`

Table 3.3: Virtual Ubuntu Elasticsearch node specifications

Hardware	**Name**
CPU	VCPU 8 Core @ 1995.307 MHz
GPU	VGPU (Cirrus Logic GD 5446)
Ram	VRAM 32 GB
Storage	SSD 40GB
Storage	HDD 1TB
Software	**Name**
OS	Ubuntu 18.04.6 LTS
Database	Elasticsearch 8.1.1

Configuration

Code listing 3.2: Backend component of the pyClipNSearchIt configuration file

```
{
    "elasticsearch": {
        "hosts": [
            "https://localhost:9200"
        ],
        "http_auth": [
            "user",
            "password"
        ],
        "verify_certs": false,
        "ssl_show_warn": false
    },
    "elasticsearch_index": {
        "number_of_shards": 9,
        "number_of_replicas": 1
    }
}
```

The Elasticsearch configuration is split into two separate parts that must be set up in a specific order for the backend to work properly. This involves first editing the *elastichsearch.yml* file present on each Elasticsearch node before starting the service for the first time. Afterwards, the second part can be configured by running *setup_backend.py* on a frontend client to finalize the rest of the cluster setup. Note that the *config.json*[5] settings displayed in listing 3.2 must have been configured correctly to allow this script to run. As for the exact Elasticsearch node configurations used in this thesis, these can be found attached in appendix D.

3.3 Dataset

During the initial stages of the research-processing phase, it quickly became apparent that much of the related work first discussed in section 2.2, had forfeited

[5]The file produced on a frontend client by running *setup_frontend.py*

using any readily available datasets for testing and/or development; that, or they at least failed to properly specify exactly which parts of the individual datasets that were used. This has made the task of later replicating their exact measurements much harder, possibly introducing a factor of bias to any presented experimental results - especially if the chosen datasets were not well vetted in the first place. Therefore, in an effort to combat the repeatability problem and ensure that similar issues will not perforate the results of *this* thesis, we have opted to use a widely known and publicly available dataset rather than creating our own.

The Multimedia Commons dataset, also known as YFCC100M, is currently the largest available multimedia collection that is free to use for anyone under the Creative Commons license (0) [45]. It consists of a total of 99 171 688 images files and 787 479 video files, all of which were originally sourced from Flickr in 2016. As a subset of Amazon's Open Data Sponsorship Program[6], the dataset is publicly available as an AWS bucket and can be downloaded using the AWS CLI client without requiring a user account. However, due to its very large size it can be somewhat difficult to download the entire dataset at once. Thus with pyClipN-SearchIt, we have instead chosen to use a smaller subset of the larger collection for all of our experiments.

Downloading the subset is done using the python script *download_dataset.py*, which runs multiple instances of the AWS CLI in parallel to download 256 hex named image folders (000 - 099), and 512 hex named video folders (000 - 1ff). Respectively these contain 387 261 images (45.5 GB) and 55 290 videos (194 GB, 1997402 seconds, 554 hours). The script takes 1 single input argument which is the output directory. A full copy of the script is available as an attachment in appendix E, but the official AWS CLI must be installed for it to be usable [46].

3.4 Experiments

Aiming to create a series of statistics that could help better showcase the thesis' end results, we ended up performing a variety of different experiments in section 4.3 to measure the actual performance of our suggested method implementation. These measurements were done in areas deemed to be crucial to the project's goal of scalability, speed and accuracy, and were done in such a way that later cross-comparisons with other approaches to video/image identification would be possible.

As a baseline, all of the conducted experiments started off using the original YFCC100M data subset described in section 3.3, but then either narrowed it down or fragmented groups of videos from it depending on the type of test that was

[6]A sponsorship program for democratizing access to high-value datasets, `https://aws.amazon.com/opendata/open-data-sponsorship-program/`

being done. Special care was taken to supply all of the utilised clips' names, as well as how they might have been potentially transformed or handled beforehand.

Chapter 4

Results

To enable a scalable approach to video indexing and search for use in the identification of fragmented and transformed image/video files, we propose in this thesis a new method/program that we have aptly dubbed "pyClipNSearchIt". This program is heavily based on research done by Cun Mu, Jun Zhao et al. in their paper "Fast and Exact Nearest Neighbor Search in Hamming Space on Full-Text Search Engines" [6], and is our attempt at iterating over previous perceptual identification approaches developed at NTNU e.g CLIPPED and PYVIDID, specifically with regards to scalability. This is not to say that other approaches and solutions have not been investigated or discussed, as they **have** in both section 2.1 and 2.2, but rather that much of the necessary information surrounding methodology and measurements is unavailable for closed source and proprietary systems such as Youtube Content ID and PhotoDNA. As such, the experimental results presented later in this chapter are mostly relevant for direct comparisons against *other* open source image/video identification systems. However first, before doing any of this, let us review the implementation of pyClipNSearchIt.

4.1 Implementation

Although Kjelsrud's PYVIDID was a substantial improvement over the original CLIPPED in many areas [1], it could in others, also be considered a downgrade. In particular, Gardåsen had originally implemented a function for threaded hashing in CLIPPED to speed up the image hashing step on larger videos and datasets [41]. However, due to some unspecified garbage collection bugs however, this feature was completely removed from the PYVIDID iteration, instead opting for a single threaded hashing approach. Thus for the sake of simplicity, the program's possible speed suffered along with its usability. Coupled with the fact that PYVIDID's sliding window matching technique is prone to search times that increase linearly with the indexed data size[1], this helped prove a motivating factor to create a better,

[1]Searches are performed by retrieving the entire indexed SQL dataset to memory, and then matching every possible query clip to every possible video by looping through all the data

faster and more scalable method of hashing, indexing and searching for videos - the result being pyClipNSearchIt.

pyClipNSearchIt is made up of a Python frontend and an Elasticsearch database backend. In contrast to CLIPPED and PYVIDID, it supports both image (.png, .jpg, .jpeg) **and** video (.mp4) searches, even going as far as to allow multiple different combinations of files to be grouped into a single index/search call at the same time. Optionally, it is also possible for the data from processed files to be outputted to a *.pregenhashes* file which can be used at another point in time. For example, it might be desirable to first run the hashing step on a powerful local host, and then defer the later indexing/searching steps to a weaker remote host. This separation of responsibility is made possible by the fact that the latter steps now mostly occur directly on the Elasticsearch nodes rather than the frontend host. In turn, this allows pyClipNSearchIt to take advantage of the horizontal scaling nature of NoSQL databases, enabling non-linear search times on potentially much larger indexed datasets.

The details on how each of the program's individual features are implemented are documented below.

4.1.1 Hashing

Of the previously mentioned content identification methods in section 2.1, we ended up choosing the already well established pHash perceptual algorithm to hash and compare transformed and fragmented images/videos. Primarily this was done to remove the dataset burden of training any potential neural identification networks[2], as well as facilitating new cutting-edge research based on traditional hemming distance comparisons. Additionally, compared with other similarity based hashing algorithms such as aHash and dHash, pHash often came out best in terms of accuracy and collision rates [15]. That being said, as the pHash algorithm only works on an image basis rather than a video basis, we needed a way to first extract the individual frames from a video before being able to process them. This led us to use the multimedia framework FFmpeg to first convert any detected .mp4 videos into multiple separate .jpeg files, and then hashing those files.

On a technical level, the program's file hashing is divided into 3 different phases to simplify development and maintenance. Namely these are: preprocessing, processing and size-reduction.

[2]Here we are referring to the fact that neural networks often require very large training datasets to have a high accuracy

Preprocessing

In the preprocessing phase the initial CLI input argument (either a file or a folder) is parsed and error checked. The program then recursively collects all of the detected images, videos and pregenhashes at the specified path, and sorts them into distinct buckets to help with later processing. This is done by checking each file's extension against predefined structures as shown in listing 4.1.

Code listing 4.1: Excerpt from src/hasher/preprocess.py

```python
def _collect(path: str) -> tuple:
    images = []
    videos = []
    pregen = []

    for root, _, files in os.walk(path):
        for file in files:
            file_type = util.get_file_type(file)
            path = os.path.join(root, file)

            if file_type == structures.FileType.IMAGE:
                images.append(path)
            elif file_type == structures.FileType.VIDEO:
                videos.append(path)
            elif file_type == structures.FileType.HASHES:
                pregen.append(path)

    return images, videos, pregen
```

Processing

One of the primary factors limiting the speed of PYVIDID's video hashing approach was the program's converter step. Kjelsrud identified this problem as FFmpeg having to save all converted video frames to disk before they could be read by PYVIDID for any further image processing. Therefore, as possible future work, he suggested that a major bottleneck could be removed if frame extraction were to take place entirely in memory instead of to-and-from disk. Although this problem certainly applies to pyClipNSearchIt too, and has been presented with a future solution in section 6.2.2, we have in our program identified and applied additional solutions that are much easier to implement, with quantifiable speed gains over earlier methods.

In a reply to a StackExchange post from 2018 [47], a user by the name of "Gyan" suggested using a FFmpeg preprocessing filter instead of the -r argument used in PYVIDID to capture a frame every *n* seconds from a video. This sounded like an interesting idea to us, and thought it worth trying to see if it made any noticeable difference to the processing speed. Luckily it did, and in testing the new approach we managed to achieve an up to 1.5x speed boost over the previously employed frame selection method, at no discernible cost to accuracy. Feeling empowered at this discovery, we tried to implement support for FFmpeg hardware acceleration, but this time with disappointing results. Notably, using the setup described in table 3.1, decoding on the GPU turned out to be much slower than on the CPU; the support therefore stayed optional and defaults to off. For the full FFmpeg launch command line used in pyClipNSearchIt see listing 4.2.

Code listing 4.2: Excerpt from src/util.py

```
def get_converter() -> str:
    system_name = platform.system()

    if system_name == 'Windows':
        converter = os.path.join(ROOT_DIR, "ffmpeg.exe")
    elif system_name == 'Linux':
        converter = "ffmpeg"

    conf = get_config()["ffmpeg"]
    hwaccel_setting = conf["use_hwaccel"]
    frame_capture_setting = conf["capture_every_n_seconds"]

    hwaccel_arg = "-c:v␣h264_cuvid␣" if hwaccel_setting else ""
    # Source: https://superuser.com/questions/1486102/fast-way-to-extract-images-
        ↪ from-video-using-ffmpeg
    # This is faster than using the -r argument because it uses a select filter
    # mod(t,3) -> every 3 seconds
    # ld(2)+1 -> output 1 frame
    filter_arg = f"-vf␣\"select='if(not(floor(mod(t,{frame_capture_setting})))*lt(
        ↪ ld(1),1),st(1,1)+st(2,n)+st(3,t));if(eq(ld(1),1)*lt(n,ld(2)+1),1,if(
        ↪ trunc(t-ld(3)),st(1,0)))'\"␣"

    return f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣{hwaccel_arg}-i␣
        ↪ \"{{}}\"␣{filter_arg}-f␣image2␣\"{{}}\""
```

After the defined number of frames every *n* seconds have been extracted from a video, the method and libraries utilized for hashing them is practically identical to the ones employed in hashing singular images. In both cases, files are read from disk in parallel using a multiprocessing pool with evenly divided "tasklets" containing smaller lists of file paths. However in an effort to ensure scalability, video tasklets stay smaller than image tasklets to allow for routine deletion of temporary images after they have been processed. This, to ensure that disk usage does not get out of hand when the program is run on larger datasets with hundreds of hours of video. Following the reading of a file, it is converted to a *Image* object using the Pillow Python library, which in turn is passed on to ImageHash's[3] pHash function to generate a 64bit hash. The length of this hash is adjustable, although smaller adjustments must be made to the Elasticsearch index mapping and ingest pipeline, as well as the python indexer and searcher to accommodate such changes. As the processor finishes, all generated hashes as well as potential timestamps are merged into a single Python dictionary named *hash_sets*.

[3]Another Python library containing many different hashing algorithms, `https://pypi.org/project/ImageHash/`

Size Reduction

A peculiar trait apparent in many videos, is that there often exists some sequences in which the camera and/or scene do not move/change over a prolonged period of time. Therefore when these frames are attempted hashed with a perceptual algorithm such as pHash, the result is that several identical hashes are produced for multiple different, albeit visually similar frames. Seen from a data analysis perspective, if the goal is strictly visual identification, this indicates the presence of possibly redundant data in the generated output. In an effort to avoid this problem, we designed two principal methods of reducing the total number of redundant indexed Elasticsearch documents:

1. **Skipping frames:** By increasing the *capture_every_n_seconds* setting in the frontend config, FFmpeg will generate less frames per video, reducing the number of identical and near-identical frames produced within a given timeframe, possibly at the cost of accuracy.
2. **Duplication removal:** pyClipNSearchIt has a built-in mechanism for ensuring per-video hash uniqueness in its internal memory structure. As seen in listing 4.3, it is possible to verify that individual video hashes stay unique at the same time as they are saved simply by treating them as keys in a Python dictionary. This removes the need for a separate post-processing step wherein duplicate hashes are detected and removed, thereby speeding up the video processing.

Code listing 4.3: Excerpt from src/hasher/process.py

```
def _p_hash_vid(files: list) -> dict:
    hash_sets = defaultdict(dict)
    _setting_frame_capture = util.get_config()["ffmpeg"]["capture_every_n_seconds"]

    for file in files:
        try:
            image = Image.open(file)
            name, frame = os.path.splitext(os.path.basename(file))
            # By using the hash as the key, we ensure that no duplicate frames are
                ↪ ever saved
            # effectively performing a uniqueness filter whilst processing!
            #
            # The set value represents the frame timestamp -> (ffmpeg output digit
                ↪  - 1) * seconds skipped (see util.get_converter())
            hash_sets[name][str(imagehash.phash(image, 8))] = (int(frame[1:]) - 1)
                ↪ * _setting_frame_capture
        except:
            pass

    return hash_sets

def _hash_videos(pool: mp.Pool, videos: list, hash_sets: dict):
    # This is taken from the middle of the function
    for result in pool.imap_unordered(_p_hash_vid, util.list_to_chunks(images, pool
        ↪ ._processes)):
        for key in result:
```

```
        existing_dict = hash_sets.get(key, None)

        if existing_dict:
            existing_dict |= result[key]
        else:
            hash_sets[key] = result[key]
```

4.1.2 Indexing

pyClipNSearchIt handles data in two similar, yet individual stages to accelerate the indexation speed of potentially large *hash_sets*[4] returned by the program's hashing phase. First, all of the videos and images that make up the hash_sets are run through a bulk generator to generate an array of Elasticsearch index documents that can be processed in parallel. Afterwards, this array is then passed on to the cluster's ingest pipeline, which represents the second stage of the indexation. In this pipeline, the ingested document hashes are split into several 4 character long keyword fields (fhash) for fast search filtering and additionally, converted to their 64bit binary representations (bhash, a signed long). The last step is necessary for performing hemming distance calculations with the *hmd64bit* script described in [6].

Listing 4.4 reveals how the first stage's document generator is set up. Note that here, the unique document *_id* is shown to be a combination of both the document's pHash and origin so that any attempts at indexing the same video/image later on will yield a detectable error message.

Code listing 4.4: Excerpt from src/database/index.py

```
def _generator(hash_sets: dict):
    for origin in hash_sets:
        data = hash_sets[origin]

        # Image
        if not type(data) is dict:
            yield {
                "_op_type": "create",
                "_index": "hashes",
                "_id": f"{origin[:16]}{data}",
                "_source": {
                    "origin": origin,
                    "hash": data,
                }
            }
        # Video
        else:
            for hash_ in data:
                yield {
                    "_op_type": "create",
                    "_index": "hashes",
```

[4]A dictionary with file origins, hashes and timestamps

```
                    "_id": f"{origin[:16]}{hash_}",
                    "_source": {
                        "origin": origin,
                        "hash": hash_,
                        "timestamp": data[hash_]
                    }
                }
```

4.1.3 Searching

To take advantage of a NoSQL database's inherent parallel processing capabilities, we needed a way to perform pHash comparisons directly inside of the Elasticsearch cluster to support fragmented and/or transformed video/image searches. This meant creating a search method that could be interpreted simply from a standard elastic query, and then returning any hits matching it upon completion. We did this by implementing a variant of the FENSHSES[5] approach briefly mentioned in section 2.1.1, fixing the encountered bugs and tailoring it to meet the specific needs of our program.

Code listing 4.5: Excerpt from src/database/search.py (query)

```
def _generate_query(hash_):
    return {
        "function_score": {
            "query": {
                "constant_score": {
                    "boost": 0,
                    "filter": {
                        "bool": {
                            "should": [
                                { "term": { "fhash.f1": hash_[0:4] } },
                                { "term": { "fhash.f2": hash_[4:8] } },
                                { "term": { "fhash.f3": hash_[8:12] } },
                                { "term": { "fhash.f4": hash_[12:16] } },
                            ]
                        }
                    }
                }
            },
            "functions": [
                {
                    "script_score": {
                        "script": {
                            "id": "hmd64bit",
                            "params": {
                                "field": "bhash",
                                # The subcode must be sent as a signed long to
                                    ↪ match the indexed long type
                                "subcode": ctypes.c_long(int(hash_, 16)).value
                            }
                        }
                    }
```

[5]Without the permutation data preparation

```
            }
        ],
        "boost_mode": "sum",
        "score_mode": "sum"
    }
}
```

Listing 4.5 helps display how our variant differs from FENSHSES's by removing the top-level *min_score* threshold, instead opting to discard weighted scores above *r* on the frontend side of things - along with any duplicate hits. This was done because the old method relied on the *hmd64bit* script returning a negative weighted score, which actually turned out to be an unsupported feature in Elasticsearch that was removed from later versions [48]. As such, with Elasticsearch **currently** not supporting an opposite *max_score* field, score thresholding on the backend turned out be infeasible.

Code listing 4.6: Excerpt from src/database/search.py (search)

```python
def run(hash_sets: dict):
    client = connect()

    for origin in hash_sets:
        data = hash_sets[origin]
        searching_for_video = type(data) is dict

        try:
            # Image
            if not searching_for_video:
                results = _search_image(client, data)
            # Video
            else:
                results = _search_video(client, data)
        except Exception as e:
            logger.error(str(e))
            continue

        logger.info("---------------------------------------")
        logger.info(f"Search␣results␣for␣'{origin}':")

        if not results:
            logger.warning("No␣similar␣matches␣were␣found")
        else:
            match = results[0]
            match_fields = match["fields"]

            match_score = int(match["_score"])
            match_origin = match_fields["origin"][0]
            match_hash = match_fields["hash"][0]
            match_timestamp = match_fields.get("timestamp", None)

            additional_info = f"␣at␣the␣timestamp␣'{str(datetime.timedelta(seconds=
                ↪ match_timestamp[0]))}'" if match_timestamp else ""

            logger.info(f"The␣best␣single␣hash␣match␣'{match_hash}'␣was␣'{
                ↪ match_origin}',{additional_info}␣with␣a␣hemming␣distance␣score␣
                ↪ of␣'{match_score}'")
            if searching_for_video:
                if (match_hash in data) and data[match_hash] != match_timestamp[0]:
                    logger.info(f"Note␣that␣local␣timestamp␣'{str(datetime.
                        ↪ timedelta(seconds=data[match_hash]))}'␣is␣not␣equal␣to␣
                        ↪ the␣found␣timestamp␣'{str(datetime.timedelta(seconds=
                        ↪ match_timestamp[0]))}',␣but␣that␣this␣is␣likely␣due␣to␣
                        ↪ both␣frames␣originating␣from␣a␣image␣sequence␣in␣the␣
                        ↪ video␣that␣is␣visually␣similar")



                if len(results) > 1:
                    best_origins = []
                    best_origins_timestamps = defaultdict(list)

                    for hit in results:
                        if int(hit["_score"]) != match_score:
                            break

                        match_fields = hit["fields"]
                        match_origin = match_fields["origin"][0]
                        match_timestamp = match_fields.get("timestamp", None)

                        if match_timestamp:
                            best_origins_timestamps[match_origin].append(
                                ↪ match_timestamp[0])

                        best_origins.append(match_origin)

                    # Count highest occurence and get seconds timestamps for those
                    best_origin, num_matches = collections.Counter(best_origins).
                        ↪ most_common(1)[0]
                    best_origin_timestamps = best_origins_timestamps[best_origin]

                    best_origin_timestamps.sort()

                    # Convert to string datetime timestamps
                    for i in range(0, len(best_origin_timestamps)):
                        best_origin_timestamps[i] = str(datetime.timedelta(seconds=
                            ↪ best_origin_timestamps[i]))

                    logger.info(f"The␣most␣likely␣origin␣is␣'{best_origin}'␣with␣'{
                        ↪ num_matches}/{len(best_origins)}'␣of␣the␣best␣scored␣
                        ↪ matches")
                    logger.info("The␣matched␣timestamps␣(in-order)␣were")
                    logger.info(best_origin_timestamps)
```

Video searches in pyClipNSearchIt are sped up using the Elasticsearch Msearch API[6] to combine multiple different image hashes into a single API request. From it, several queries can be concurrently read and processed by the cluster, keeping the overall search time low for both shorter and longer video clips. From the result, the most likely origin is chosen by counting the highest number of origin occurrences with the lowest hemming distance scores; shown in listing 4.6.

[6] Specifically we are using the Elasticsearch Python client's msearch function, `https://elasticsearch-py.readthedocs.io/en/v8.2.0/api.html?highlight=msearch#elasticsearch.Elasticsearch.msearch`

4.2 Source Code

The full source code of pyClipNSearchIt is available publicly as a Git repository on Github under the user WilhelmThon [49]. For potential future work or pull requests, refer to chapter 6.

4.3 Experiments

An essential component of the scientific method is confirming whether a given hypothesis is actually correct, or not. To do this, we define a series of experiments which the hypothesis can be tested on, and then later, verified by comparing the *produced* experimental results against the *expected* results. If the former turns out to disagree with the latter, we know as Richard Feynman once stated in one of his lectures [50], that our initial assumptions and results must be wrong and that the hypothesis should be adjusted or rejected thereafter.

For pyClipNSearchIt, we developed three different types of experiments and hypotheses to test the scalability and reliability of the program:

1. **Size Reduction** The program is able to reduce the general indexed dataset size
2. **Speed** The program is able to provide fast hashing, indexation and searching of videos/images
3. **Accuracy** The program returns temporally and content accurate search results

4.3.1 Size Reduction

With very large datasets, a sizeable portion of hardware storage is almost always required to store the necessary amount of data permanently to disk[7]. Naturally, this brings to the table a discussion surrounding what the actual cost implications of running a NoSQL or SQL type of database are, and correspondingly, how these costs can best be mitigated through means of smart hardware and/or software solutions. In the case of our program, we opted to go for the latter option by reducing the initial dataset size on the frontend **before** sending it to the Elasticsearch cluster's ingest and data nodes for storage.

To measure the efficiency of our approach, we used the *video* part of our dataset and ran the exact same type of test three times to count the number of hashes produced by pyClipNSearchIt at varying size reduction settings. Measurements were done by running the Python script *count_hashes.py* attached in appendix F on the .pregenhashes files output by the program. Below is list of all the settings

[7]Here, we refer to Hardware disk drives (HDD) and Solid state drives (SDD)

that were used with the corresponding test results being shown Figure 4.1.

a) Original number of hashes in video dataset, w/ duplicate hashes
b) pyClipNSearchIt output, capture_every_n_seconds=1 w/o duplicate hashes
c) pyClipNSearchIt output, capture_every_n_seconds=3 w/o duplicate hashes

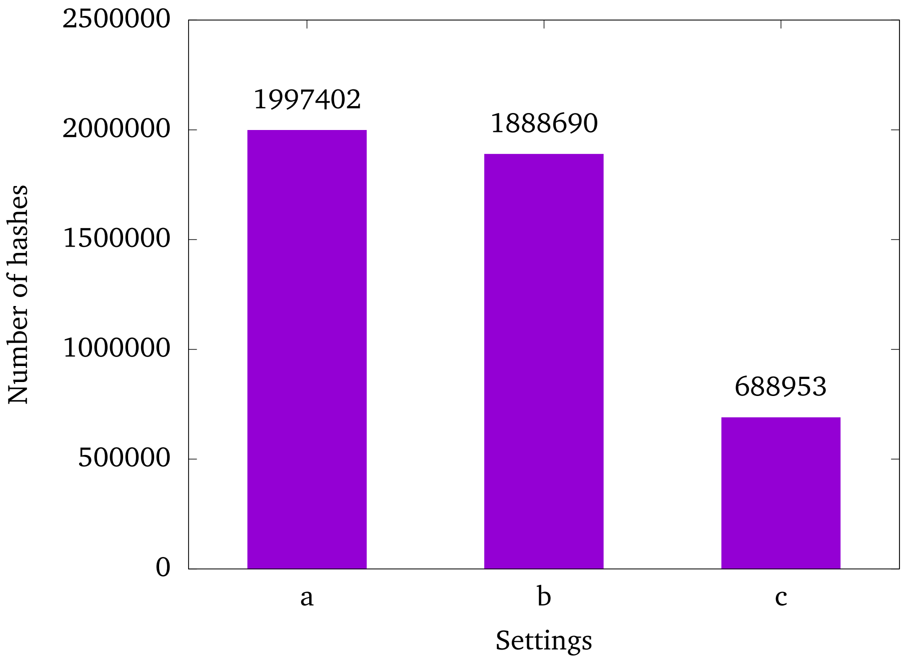


Figure 4.1: The number of hashes produced at different pyClipNSearchIt settings

4.3.2 Speed

A key trait that characterises a fast and scalable computer system, is its ability to readily process and handle large amounts of data within a relatively short period of time. For a video and image identification system such as pyClipNSearchIt, this meant measuring the speed at which the program was able to hash a given user input, and correspondingly, how fast this input was then able to later be indexed or searched. To perform the relevant measurements, we utilized the *time_ns()* method from the Python *time* module and generated a total of 10 timestamps for each of the individual tested components. The mean average of these is what has been presented in the graphs below.

Listings 4.7, 4.8 and 4.9 show the numerous locations at which the *time_ns()* methods were inserted for all of our tests.

Hashing

Code listing 4.7: Time measurement in src/hasher/__init__.py

```
    logger.info("Generating␣hashes,␣please␣wait...")
    time_before = time_ns()
    hash_sets = process.run(images, videos)
    time_after = time_ns()
    time_result = time_after - time_before
    logger.info("Finised␣generating␣hashes!")
```

The speed of the hashing component was tested using 3 separate videos of varying lengths ranging from 11 seconds to 31 minutes of runtime. This made it easy to visualize how the program handled longer versus shorter videos, and how adjusting frontend settings such as the *capture_every_n_seconds* could impact the hashing speed both negatively and positively.

Below is a list of the exact videos that we used, along with the corresponding test results; shown in figure 4.2.

a) 16869d82c0513d34dfa19bf41e75883.mp4, 11 seconds
b) 168cef441261f67574855bbade76a9e.mp4, 3 minutes
c) 13cdd7751b3efe83622d412b4a1ca141.mp4, 31 minutes and 27 seconds

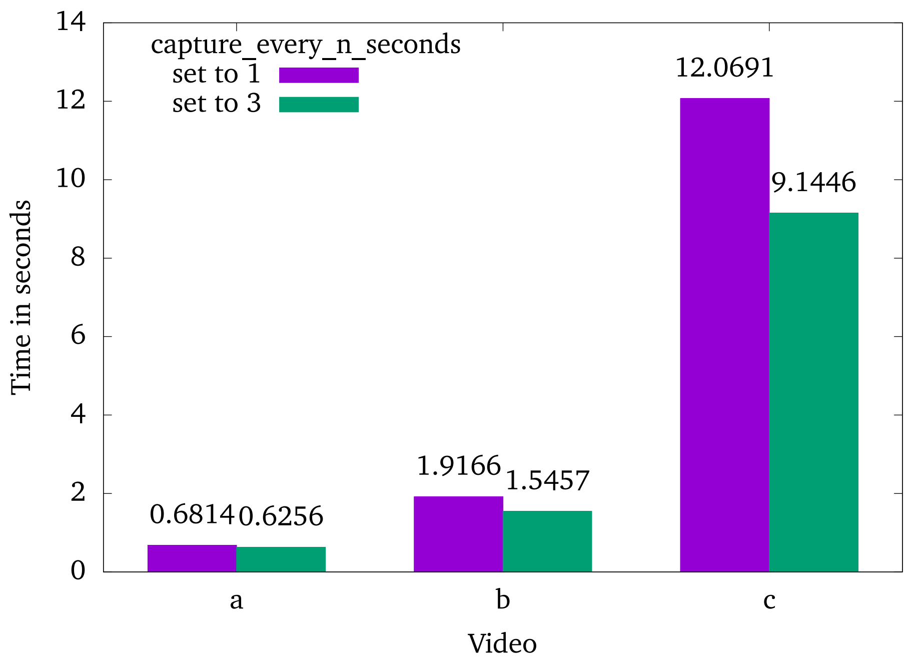


Figure 4.2: The mean average measurements from hashing a single video 10 times

Indexing

Code listing 4.8: Time measurement in src/database/__init__.py (indexing)

```
logger.info("Indexing␣hashes,␣please␣wait...")
time_before = time_ns()
index.run(hash_sets)
time_after = time_ns()
time_result = time_after - time_before
logger.info("Finished␣indexing␣hashes!")
```

To test the indexing speed of our program, we wanted to see how fast a dataset containing 387 261 hashes could be indexed compared to one containing e.g. 1 888 690 hashes. As such, we ended up defining three subsets of data from the larger YFCC100M dataset described in section 3.3, which we repeatedly inserted into our Elasticsearch database one at a time to test variations in speed; each time making sure to drop the backing index before conducting a new test. In doing so, we achieved the experimental results that are presented in figure 4.3.

a) image dataset, 387261 hashes
b) video dataset with capture_every_n_seconds=3, 688953 hashes
c) video dataset with capture_every_n_seconds=1, 1888690 hashes

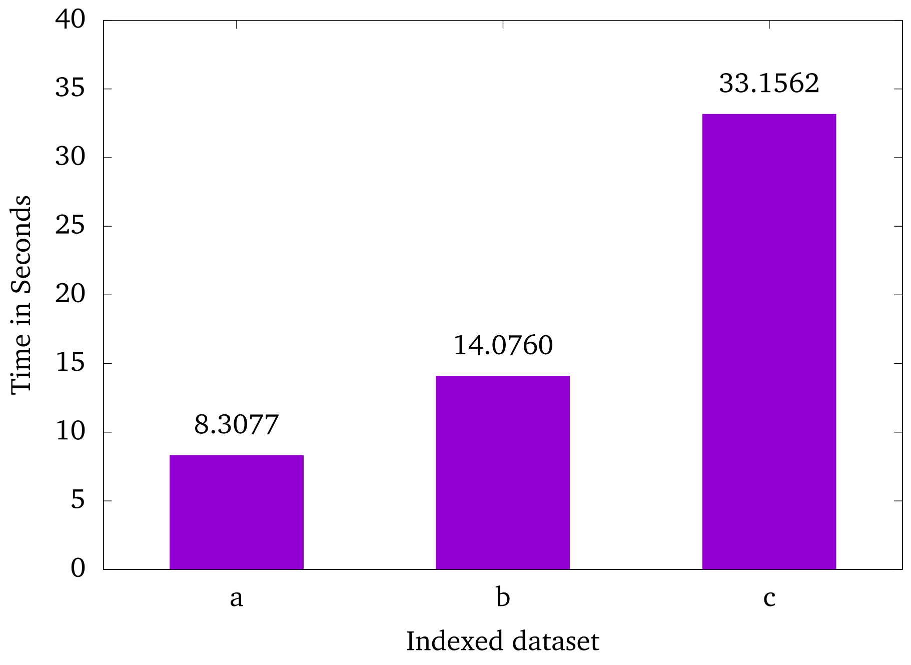


Figure 4.3: The mean average measurements from indexing a dataset 10 times

Searching

Code listing 4.9: Time measurement in src/database/__init__.py (search)

```
    logger.info("Searching␣for␣matches,␣please␣wait...")
    time_before = time_ns()
    search.run(hash_sets)
    time_after = time_ns()
    time_result = time_after - time_before
    logger.info("Finished␣search!")
```

In line with our goal of providing a fast and scalable approach for video searches, one of the benchmarks that we created to measure pyClipNSearchIt's performance, was how fast it could identify a single video inside a dataset comprised of thousands of videos and images. In effect, this helped gauge the cost-benefit of the new in-query search method described in section 4.1.3 versus the old local in-memory search method employed in PYVIDID and CLIPPED.

For the experiment we used the input files enumerated in the list below and plotted their respective results into figure 4.4.

a) 16869d82c0513d34dfa19bf41e75883.mp4, 11 seconds, capture_every_n_seconds=1
b) 168cef441261f67574855bbade76a9e.mp4, 3 minutes, capture_every_n_seconds=1
c) 13cdd7751b3efe83622d412b4a1ca141.mp4, 31 minutes and 27 seconds, capture_every_n_seconds=1

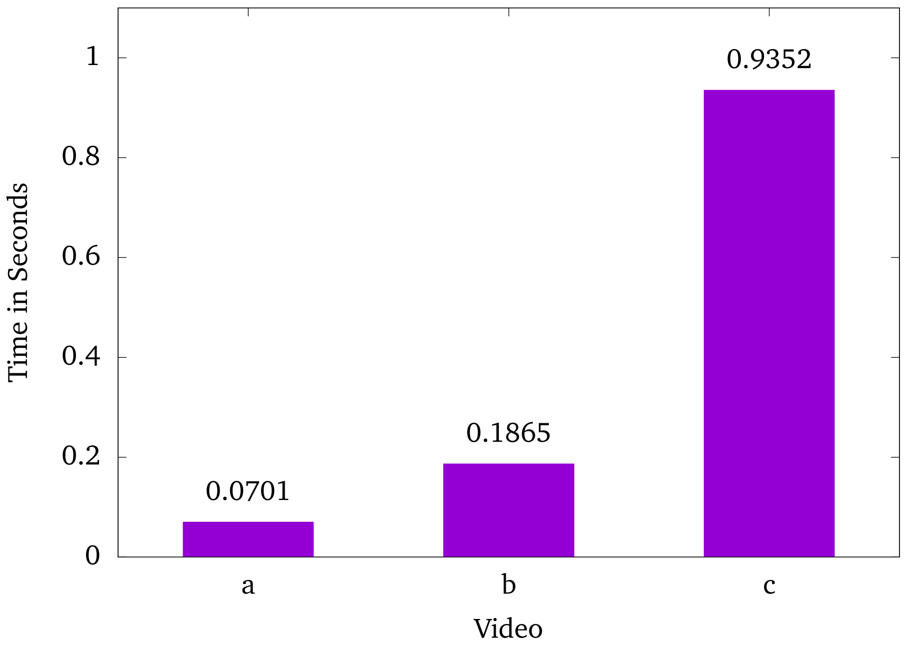


Figure 4.4: The mean average measurements from searching for a specific video 10 times in the combined video + image dataset

4.3.3 Accuracy

Arguably one of the most important metrics by which a system intended for video identification can be measured is its accuracy. Specifically, this refers to how consistent the system is at returning the correct origin (filename) for any given set of search inputs (images/videos); precondition being that a variant of the data is already stored in the database. As such, a high degree of accuracy will indicate that a video identification system is *reliable*, whilst a low accuracy on the other hand, will indicate that it is *unreliable*.

To determine which of these classifications was most applicable in the case of pyClipNSearchIt, we conducted two independent experiments to measure the system's ability to accurately identify both fragmented and transformed videos inside a larger dataset consisting of images and videos. The following list describes all of the settings used at each step in the testing process:

a) r=5, capture_every_n_seconds=1 indexed dataset and query clip
b) r=5, capture_every_n_seconds=3 indexed dataset and =1 for query clip
c) r=10, capture_every_n_seconds=1 indexed dataset and query clip
d) r=10, capture_every_n_seconds=3 indexed dataset and =1 for query clip
e) r=20 capture_every_n_seconds=1 indexed dataset and query clip
f) r=20, capture_every_n_seconds=3 indexed dataset and =1 for query clip

Fragmented

Using the *extract_clips.py* script attached in appendix G, we extracted 3 sets of clips from the *101* video dataset subfolder (196 videos) to test how accurately our program could identify fragmented clips (5 seconds, 30 seconds, full length) in longer indexed videos. We display the results from these tests in table 4.1.

Table 4.1: The accuracy for different types of fragmented clip searches

Clip length	Matches - a	Matches - b	Matches - c	Matches - d	Matches - e	Matches - f
5 seconds	100% (177/177)	61% (108/177)	100% (177/177)	75% (133/177)	100% (177/177)	75% (133/177)
30 seconds	100% (77/77)	95% (73/77)	100% (77/77)	94% (75/77)	100% (77/77)	94% (75/77)
Full length	100% (196/196)	99% (194/196)	100% (196/196)	99% (194/196)	100% (196/196)	99% (194/196)

Transformed

We tested 6 different transformations in total to determine pyClipNSearchIt's skill at identifying transformed video files. Table 4.2 shows which types of transformations these were, along with the relevant result accuracy metrics. For the query clips, we chose 13 videos from the *012* video dataset subfolder, and then ran them through the *transform_videos.py* script attached in appendix H to apply any necessary transforms. The clip lengths varied from 20 seconds to 1 minute.

Table 4.2: The accuracy for different types of transformed full clip searches

Transform	Matches - a	Matches - b	Matches - c	Matches - d	Matches - e	Matches - f
Brightness +25%	100% (13/13)	85% (11/13)	100% (13/13)	100% (13/13)	100% (13/13)	100% (13/13)
Brightness -25%	100% (13/13)	77% (10/13)	100% (13/13)	92% (12/13)	92% (12/13)	92% (12/13)
Contrast +25%	100% (13/13)	100% (13/13)	100% (13/13)	100% (13/13)	100% (13/13)	100% (13/13)
Contrast -25%	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)
Rotate 10° clockwise	0% (0/13)	0% (0/13)	15% (2/13)	8% (1/13)	15% (2/13)	8% (1/13)
Crop +25%	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)	0% (0/13)

Chapter 5

Discussion

Having finished presenting the results of our thesis, we can now move on to discussing their potential implications, as well as how they specifically relate to the research questions stated in chapter 1. To answer the questions, we will first go through all of them individually, referring to them by their acronyms, and then draw parallels to sections of useful information or experiments providing relevant context and background. As an implied necessity, this also means comparing how our solution performs against others, and what the most suitable applications for taking advantage of the demonstrated strengths are.

One of the first research questions we set out to answer were R.1 and R.2. In a sense, the answers to both were inherently closely connected, sharing the common goal of discovering how *previous* techniques and solutions were designed and implemented. This in turn, carried an effect on the general design philosophy concerning our *new* method, as we attempted to work around and find solutions for existing flaws and problems. For example, of the numerous content identification techniques we researched, signature generation was found to be the most simple and cost effective method for creating a scalable video identification system. This, as the deep machine learning approach required too large of a front-up investment in terms of training and implementation details. The simplicity of PYVIDID's hemming distance comparison approach also proved alluring, combining with FENSHES's Elasticsearch in-query processing to create a simple to understand, yet highly scalable, fast and accurate solution.

Below, we demonstrate how pyClipNSearchIt's image and video identification method substantiates an improvement over Ola Kjelsrud's earlier PYVIDID approach for large datasets, particularly in regards to its size reduction (R.3) and speed (R.4).

5.1 Experiments - Size Reduction

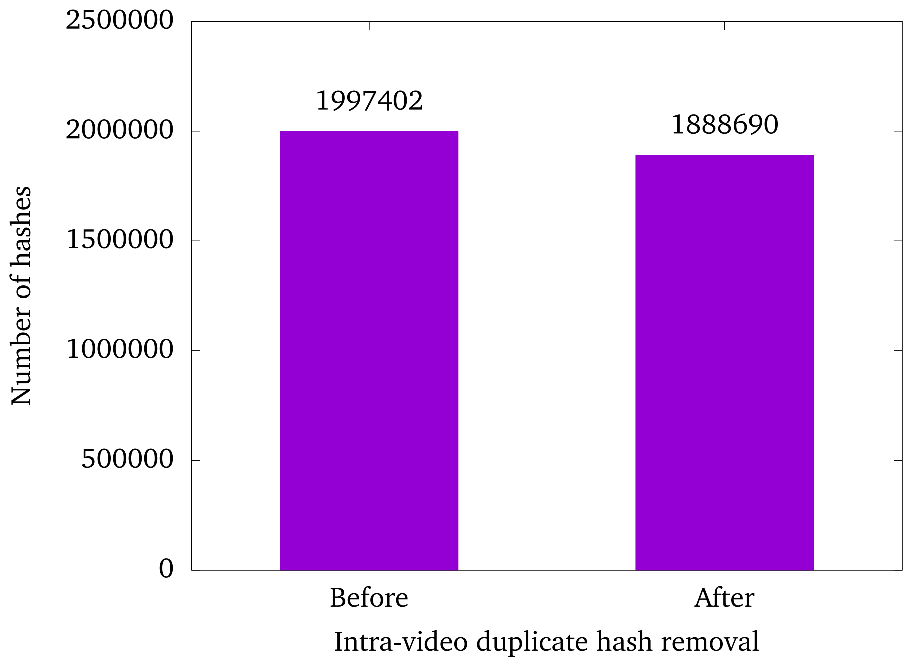


Figure 5.1: The number of total generated hashes from the video dataset before and after applying intra-video duplicate hash removal

Though pyClipNSearchIt's method of storing perceptual hashes as Elasticsearch documents instead of SQL rows results in a larger data footprint when compared to PYVIDID, the **amount** of indexed hashes is actually smaller. This, without having any observable effect on the program's measured search accuracy. By applying the intra-video duplicate hash removal method described in section 4.1.1, we managed to achieve an approximate 5.44% reduction in the total number of generated hashes by pyClipNSearchIt; shown in figure 5.1. Although this result might appear inconsequential first, it is important to remember that it can vary drastically depending on the indexed videos' size and type, i.e. videos with little movement in them will contain more duplicate hashes and therefore display a higher size reduction. In any case, the number of hashes that need to be processed for every image/video search remains consistently reduced for most longer videos. Arguably, this possibility for faster processing times is also a better metric for scalability than raw data size, as CPU power is a considerably more valuable resource than permanent disk storage.

It is possible to realize even larger size reductions by increasing the *capture_every_n_seconds* hash setting to skip video frames as shown in figure 4.1. However, this might result in a lower search accuracy and as such, should be made in consideration of our experimental results from table 4.1. That being said, we have with this discussion proven our size reduction hypothesis from section 4.3 and answered research question R.3.

5.2 Experiments - Speed

From the experimental results in section 4.3.2, we can with a high degree of certainty declare that our speed increase hypothesis was indeed correct. As to exactly how, and on the topic of research question R.4, we refer to the ensuing discussions below pertaining to each individual test result.

5.2.1 Hashing

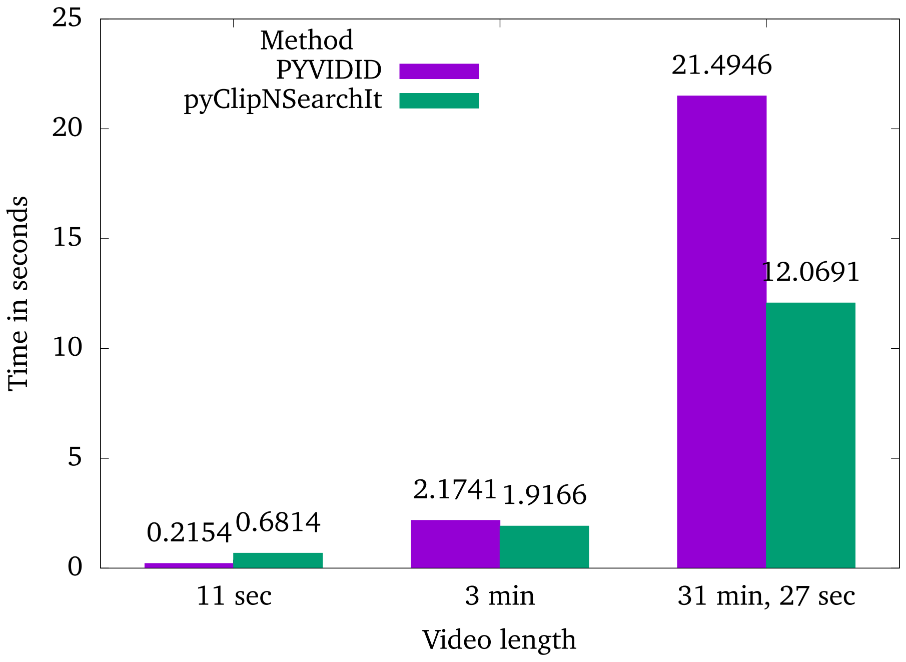


Figure 5.2: The mean average measurements from hashing a single video 10 times compared to PYVIDID

Utilising a preprocess FFmpeg select filter for extracting image frames instead of the normal *r* argument used by PYVIDID, pyClipNSearchIt saw an up to 78% speed increase over the previous approach. Shown in figure 5.2, this speed up is largely dependent on the given input video's duration, wherein the performance disparity gap will grow accordingly with the measured length of the runtime. In short: our method best displays its superiority on longer types of videos. The only area in which our method may be worse is if the initial input duration is below 3 minutes, thus causing the filter initialization costs to offset all of the frame select benefits.

Though, this does not imply that our approach is **always** slower for shorter videos, as our image hashing phase[1] excels at handling collections of videos rather than single file inputs; due to its multithreaded design.

As for how different *capture_every_n_seconds* configurations might impact the processing speed, we observed in our tests that lower settings yield slower speeds - although not at a rate which one might be expecting. The experimental results plotted in figure 4.2 show that though the number of extracted frames is 3 times higher at a setting of 1 compared to a setting of 3, the processing speed does not increase linearly between them. Therefore, we suspect that the actual number of frames extracted by the select filter is not the main bottleneck in FFmpeg.

5.2.2 Indexing

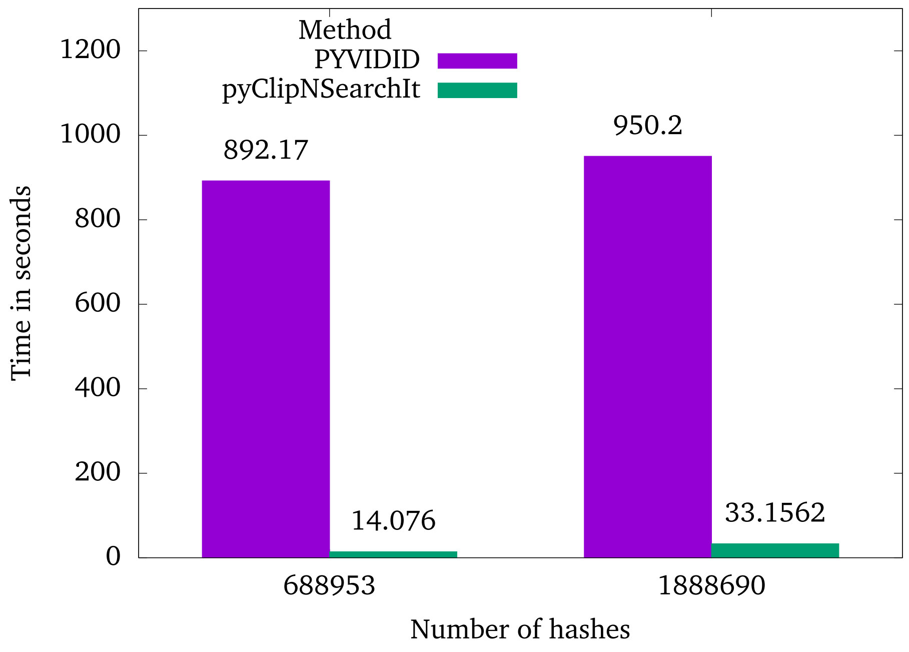


Figure 5.3: The mean average measurements from indexing a video dataset 10 times compared to PYVIDID

Parallelizing the indexing operation through a combination of multithreading, bulk queries and ingest pipelines has led to what can be described as nothing short of a substantial leap in terms of speed and scalability. Processing a large dataset consisting of 55 290 videos, pyClipNSearchIt is at its best 6238% faster than PYVIDID at indexation. Notably, this is also using a linux frontend host with only 2 virtual CPUs for testing our program, compared to testing Kjelsrud's on a

[1]Note: not the FFmpeg image frame extraction

windows frontend host with a dedicated 6 core Intel i5-9600K CPU. Therefore, it is theoretically possible to achieve an even greater disparity in performance by scaling up the number of cores on the frontend host communicating with the Elasticsearch cluster.

Figure 5.3 shows that the major bottleneck for PYVIDID is not the number of *hashes* being indexed, but rather, the number of *videos* that are processed. This can be explained by the fact that the program is designed to spread its indexed data across two tables (VIDEO_NAMES, VIDEO_HASHES), substantiating the need for separate SQL insertion statements. Furthermore, all of these statements are single inserts instead of bulk inserts, creating a constant back-and-forth conversation between the frontend and the backend database. Our method in contrast, divides the processed videos and images across a number of different threads which asynchronously send bulk queries to the Elasticsearch ingest pipeline. Thus, as seen in figure 4.3, the speed of indexation does not increase linearly with the number of hashes or videos/images, and scales well for large datasets; made evident by example a) to b) constituting a 5x increase in the number of hashes, but only a 4x increase in processing time.

5.2.3 Searching

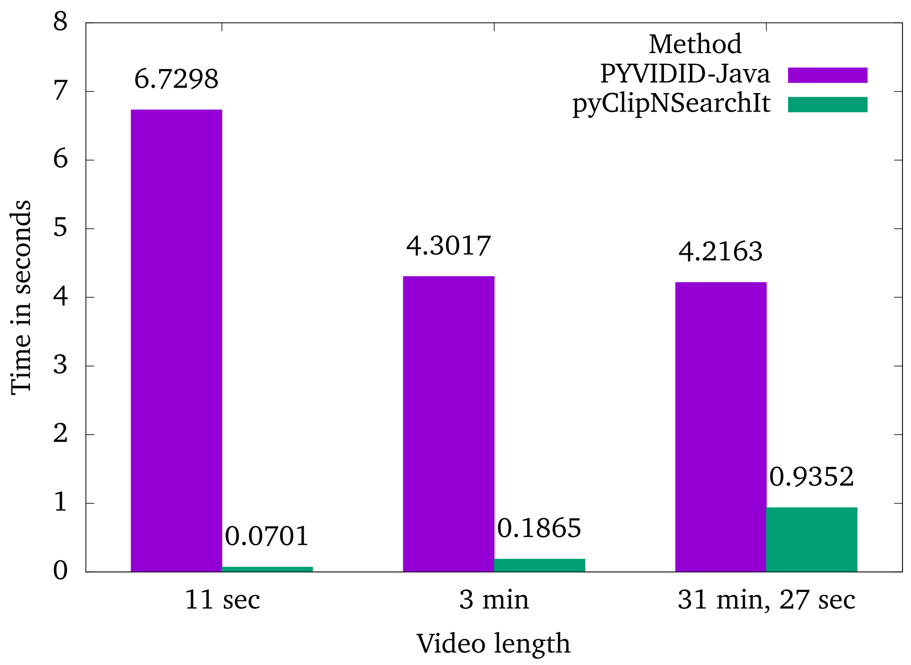


Figure 5.4: The mean average measurements from searching for a specific video 10 times in the combined video + image dataset compared to PYVIDID

In his thesis, Kjelsrud presented PYVIDID as a highly performant alternative to CLIPPED, offering sub-second match times for both short (30 seconds) and long (25 minutes) video searches. In the process of reproducing his experimental results for our own comparisons though, we discovered the latter claim to most likely be highly exaggerated, at least when applied to very large datasets. The basis for this assumption is that Kjelsrud only ever opted to test his program on a small, yet lengthy selection of videos. Thus, the same type of SQL query problem highlighted in section 5.2.2 for indexing managed to become equally detrimental to the search performance. That is, according to our analysis of the underlying Java source code. Consequently, ranked against pyClipNSearchIt, PYVIDID performed very poorly against our test dataset[2], as exemplified in figure 5.4.

Ironically, as the search time for longer videos increases with our Elasticsearch in-query comparisons, the opposite is true for PYVIDID's in-memory comparisons. This phenomenon can be explained by the number of short videos in the dataset far outnumbering the number of long ones, proving advantageous for PYVIDID's search method which skips comparisons if the indexed video is shorter than the input clip. However in the end, pyClipNSearchIt's search speed still measures around 96 to 4.5 times faster than PYVIDID's and remains consistent even if the quantity of indexed data grows.

5.3 Experiments - Accuracy

Analysing the results from section 4.3.3, we can confirm that our stated accuracy hypothesis is positively true for the fragmented experiments, but less so for the transformed experiments. This mainly boils down to limitations with the used perceptual hashing algorithm as we will explain later in this section.

5.3.1 Fragmented

Table 5.1: The best measured accuracy for fragmented clip searches compared to PYVIDID

Clip length	Matches - PYVIDID	Matches (c) - pyClipNSearchIt
5 seconds	85% (151/177)	100% (177/177)
30 seconds	53% (41/77)	100% (77/77)
Full length	85% (167/196)	100% (196/196)

pyClipNSearchIt's fragmented search accuracy can vary greatly depending on a query clip's duration, as well the used indexing and thresholding settings. This is best exhibited in the tests a), b) and d) listed in table 4.1, showing that if the

[2]Note that we did not index any images in PYVIDID's database for the comparison, as the program does not support them

indexed dataset uses a high *capture_every_n_seconds* setting, the search hemming distance threshold *r* must also be raised to compensate for possible discrepancies in the query clip's perceptual hashes versus the index ones. This, owing to the compared hashes possibly being generated from visually dissimilar frames, impacting both the match score and accuracy negatively. For lower frame capture settings though, the *r* threshold may safely be kept small whilst still retaining an overall 100% match accuracy for non-transformed clips.

Compared to PYVIDID, our search method is definitely the one that comes out on top with regards to fragmented search accuracy. Whilst the former at best returns 53%-85% accurate results for most videos, ours manages to return 100% accurate results for **all** videos using a *capture_every_n_seconds* setting of 1; the comparison being shown in table 5.1. We believe the most simple explanation for this difference, is that the Java implementation of PYVIDID's sliding windows matching algorithm is bugged. Specifically the offending line is thought to be `if(matchhashes.size() <= dbhashes.size()-j+1)`, which causes all hash comparisons but the first to be skipped in each video loop iteration.

5.3.2 Transformed

Table 5.2: The best measured accuracy for different types of transformed full clip searches compared to PYVIDID

Transform	Matches - PYVIDID	Matches (c) - pyClipNSearchIt
Brightness +25%	77% (10/13)	100% (13/13)
Brightness -25%	92% (12/13)	100% (13/13)
Contrast +25%	69% (9/13)	100% (13/13)
Contrast -25%	0% (0/13)	0% (0/13)
Rotate 10° clockwise	46% (6/13)	15% (2/13)
Crop +25%	54% (7/13)	0% (0/13)

In testing how various different video transformations would effect the accuracy of our search results, we noticed a familiar problem that has many times before, plagued visual identification systems utilising a fuzzy hashing algorithm such as pHash. In essence, since the perceptual hashes have to be noticeably similar to draw meaningful comparisons, both the image shape and color must stay predominantly consistent across any user applied transformations. Thus, changes heavily affecting these identifying traits such as contrast shifts, rotations and cropping can damage the system's overall experienced reliability. Table 4.2 showcase this problem perfectly, where we observe that although brightness transformations have no observable effect on the search accuracy, rotations do. Therefore as a countermeasure, we increased the hemming distance threshold *r* between test a) and c) to allow for greater distances between compared frame hashes to increase the general accuracy.

Most notably, transformed video searches is the only area in which PYVIDID has an edge over pyClipNSearchIt. This is most probably due to the per. video window sliding technique calculating a better average similarity score than our "all-at-once" approach. For a complete accuracy comparison between the two methods we refer to table 5.2.

5.4 Potential Applications

With the ability at which our new method of performing image and video identification can scale, there exists a wide variety of exciting applications and use-cases whose requirements are now met and made possible by pyClipNSearchIt. This does not however, imply that **no** corresponding previous solutions exist, but rather that these solutions are either a) *hard-to-implement*, b) *licensed* or c) *proprietary* as was discussed earlier in section 2.2. Therefore the listed applications below should primarily be viewed in the light of the open source nature of our approach, and judged thereafter.

5.4.1 Video Source Finder

Considering the popularity of the audio identification service Shazam, it is somewhat of a wonder that no similar alternative yet exists for identifying video sources. On several occasions we managed to find users that had failed to track down the origin of a particular movie scene they had seen or downloaded, and as a consequence complained about it on a random internet forum. Knowing this, along with the fact that there is a readily addressable market for an open video source identification service, our scalable method could potentially prove the perfect fit for the application.

5.4.2 Forensic Investigations

Similar to GriffEye Analyze, pyClipNSearchIt's image and video identification method might prove useful for identifying victims of child abuse and pornography in the setting of professional forensic investigations. Both systems support data sharing[3] across teams to encourage efficient cooperation, and both are designed from the ground up to meet the big data processing requirements of today. As such, with the added bonus of our system being both open source and free-to-use, pyClipNSearchIt is not just a viable alternative to present forensic solutions, but also a cost effective one.

[3]Using the Elasticsearch migration and snapshot features, actors can easily share data indexes amongst themselves, `https://www.elastic.co/guide/en/cloud/current/ec-migrating-data.html#ec-migrating-data`

5.4.3 Copyright Enforcement

A very common use-case for image and video identification systems is to help with copyright enforcement. Instead of manually having to review reported content on a case by case basis, it is much easier to simply feed it through a matching algorithm to detect if the copyright has potentially been infringed upon. Most popularly, this has for many copyright holders meant using Youtube's Content ID system, allowing them to block or monetize any videos on the platform containing their copyrighted content. However, being that this system is solely limited to just Youtube, it is desirable to have another solution such as pyClipNSearchIt to detect infringements on other platforms as well.

Chapter 6

Future Work

Although the first experimental results from testing pyClipNSearchIt show a definitive improvement over earlier available methods for fragmented and transformed images/video identification (especially with regards to scalability), there still remains a lot of areas in which the program may still yet be refined or reworked. If lucky, some of these adjustments might just simply require a few lines of extra code to be successfully implemented, whilst others may require larger and more fundamental changes to be made to the codebase; perhaps even going as far as porting the entire project to another programming language entirely. Therefore, to simplify the task of improving pyClipNSearchIt for future researchers or collaborators, suggested areas of future work have been separated into two different categories: Low-hanging fruit and High-hanging fruit. The first category is intended for changes that can be classified as easy-to-implement but still of moderate value, most likely not already having been implemented due to a lack of time or necessity. In contrast, the second and last category is intended for more major and/or breaking changes to the underlying core concepts which will require an in-depth knowledge of how the program works to be implemented successfully. Thus, for the especially interested, suggestions in the latter category might be thought of as core ideas for potential future papers, possibly granting very high yields in observable factors such as speed, accuracy and data size. Note however, that pyClipNSearchIt is licensed under the GPL 3.0 license [51].

6.1 Low-hanging Fruit

6.1.1 Additional file metadata

Whilst the current iteration of the program *does* index *some* basic metadata from processed files, namely names, types and video frame timestamps, there still remains leftover metadata that can prove useful in specialized applications such as forensic investigations and image-document identification. Particularly, the created, modified and accessed timestamps as well as file size are useful for tracking the age and relevancy of indexed files. Unfortunately, small but non-backwardscompatible changes must be made to the current storage format to save and index this additional data. Though, this might be a small price to pay for additional valuable information.

6.1.2 Native phash functionality

Currently two external python libraries, Pillow and ImageHash, are used in conjunction to generate the pHashes used by pyClipNSearchIt for indexing and searching. This is less than ideal considering the overhead presented by their corresponding implementations, which in ImageHash's case means a heavy reliance on python lists and general ineffective memory usage. Consequentially, this has the effect of incurring a non-negligible speed penalty on the program's hashing step, which slows down the processing of larger datasets. As such, a direct implementation of pHash in an independent c/c++ python module would likely return significant results in terms of improving the hash processing speed.

6.1.3 Greater size-reduction

It is possible to further reduce the size of indexed data by implementing a post-processing step to remove *nearly* identical hashes present in the processed videos. By comparing the generated hashes against each other on a per-video basis with a set hemming distance threshold t, groups will naturally form in which the members' hash similarity lies below the threshold. Then, from these groups, only a single member needs to be retained for the indexation step - optionally with the other members' timestamps appended. Though, know that this will most likely impact the match accuracy of searches, and only works if the hemming distance search threshold r is higher than or equal to the post-processing threshold t.

6.1.4 Detection of temporal continuity

Given that the program already returns matched frame timestamps for video searches in a sorted order, it should also be possible to highlight temporal continuity based on a set time threshold. For example, if three timestamps are evenly split 3 seconds apart, it would indicate a continuity of 6 seconds of footage. Highlighting this in the final output would make it easier for a user to determine if the query video is a single continuous clip from the original video, or, a montage of multiple.

6.1.5 Graphical user interface

A graphical user interface would help make pyClipNSearchIt much more accessible by lowering the barrier of entry for less tech-savyy users. Additionally it would serve as an important step towards improving production-readiness for any potential future applications described in section 5.4. The most straight forward implementation would be to use an existing Python framework such as wxPython[1] to present the program's functionality in an easy-to-use GUI shell, however a javascript wrapper could also work if the desire is to expose user input and output directly in a web-interface.

6.1.6 Better error handling

Despite there already being extensive error checking in-place for argument parsing and path handling, there are still some cases where the program might fail if files are corrupted or the database search query times-out. To better handle these errors, it would therefore make sense to check a file's magic number [52] in addition to its extension to ensure that the correct file format is being parsed, and re-try any failed search queries *n* set of times.

6.2 High-hanging Fruit

6.2.1 C/C++ port

Porting the source code of pyClipNSearchIt to a compiled language such as C or C++ instead of a dynamic language like Python, would yield substantial speed improvements over today's current version. This is due to the interpreted and allocation heavy nature of Python, along with the caveats it carries surrounding multi-threading and function calls. As an example, true threading in Python requires using the multiprocessing module to avoid hitting the global interpreter lock, which creates large overheads for sharing data between processes. Furthermore, it is not possible to change strings in-place without creating any copies which is bad for heavy string operations. Luckily most of these grievances are not really problems in the suggested language ports, with added possibilities for arena memory allocators and string views; at the added cost of potential new vulnerabilities if not careful. Nevertheless, porting the project is a task for the special enthusiast.

6.2.2 Native FFmpeg functionality

Because reading and writing files from/to disk is substantially slower than handling them in memory, using the FFmpeg *libavcodec* c library rather than opting to run a separate process would most likely provide a significant speed boost over the current video frame extraction method [53]. Either this can be thought implemented as part of the previously mentioned C/C++ port of the project, or as a smaller c module library called directly from Python. This is similar to how a FFmpeg process is launched through the subprocess module today. As an added

[1]wxPython is a cross-platform tookit for creating graphical user interfaces in the Python programming language, `https://www.wxpython.org/pages/overview/`

bonus, this would also circumvent the need to keep track of and clean FFmpeg extraction remnants every 20 or so video passes.

6.2.3 Better perceptual hashing algorithm

As it stands, the perceptual hashing algorithm utilized in the thesis, pHash, does not really deal well with differentiating larger blocks of colors due to its linear color averaging approach. This can impact the overall accuracy of any retrieved search results and should thus be seen as a shortcoming with the algorithm. To work around this issue, a new open-source perceptual hashing algorithm with comparable metrics to pHash should therefore be attempted found or created to lessen any potential false positives. Notably, there already exists some alternatives to pHash such as PhotoDNA by Microsoft, although both it and several other algorithms are locked under proprietary licenses rendering them unusable in open source projects.

Chapter 7

Conclusion

In this master's thesis we have discussed the challenge of scalability, as well as how it relates to our topic of video identification. We highlighted that many of the existing video identification systems today struggle with ever increasing data volumes, and how none of them really appear viable for use in actual production settings. This is a consequence of them either being too slow to operate in real time on big data, or simply classifying as proprietary in nature and thus restricting their potential public adoption. Therefore, in response we set out to create a more open and scalable approach to performing video identification, which could be used to efficiently identify both fragmented and transformed image/video files in very large datasets; the end result of this work being pyClipNSearchIt.

pyClipNSearchIt is a highly performant and scalable video identification method that is specifically tailored for big data applications. It is heavily inspired by PYVIDID, which uses a perceptual hashing algorithm - pHash - and hemming distance comparisons to accurately identify video clips within a pre-indexed dataset. However in contrast to PYVIDID's in-memory sliding window matching technique, our approach instead leverages the new FENSHES in-query technique to perform hash comparisons directly inside of a ElastichSearch cluster's data nodes. This speeds up the search times by approximately 96 to 4.5 times over the previous approach, when tested on a dataset consisting of 55 290 videos and 387 261 images. This measurement of course, will vary depending on the given input clip's runtime duration.

As another achievement, pyClipNSearchIt is also able to easily identify and remove intra-video duplicate hashes at no additional cost to the program's hashing step. This helps reduce the total number of generated frame hashes by around 5.44% and decreases the overall size of the indexed dataset. For the data indexation process itself, we managed (through the application of multithreading, bulk indexing APIs and ElasticSearch ingest pipelines) to achieve a substantial 6238% speed improvement over PYVIDID. This just goes to demonstrate the improved scalability and viability of our new approach compared to earlier ones.

We believe that the most suitable application of pyClipNSearchIt is as a video alternative to the audio identification service Shazam - However, pyClipNSearchIt stands equally tall on its own as yet another stepping stone in the pursuit of providing knowledge for a better world; as is NTNU's official motto.

Bibliography

[1] O. Kjelsrud, 'Using perceptual hash algorithms to identify fragmented and transformed video files,' Gjøvik University College, 2014.

[2] B. Berisha and B. Mëziu, *Big data analytics in cloud computing: An overview*, Feb. 2021. DOI: `10.13140/RG.2.2.26606.95048`.

[3] Statista, *Youtube: Hours of video uploaded every minute 2020*, `https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/`, (Accessed on 20/04/2022), Feb. 2020.

[4] Griffeye Technologies, *Griffeye analyze | an open and modular software platform for your digital media investigations*, `https://www.griffeye.com/`, (Accessed on 19/04/2022), Apr. 2022.

[5] Microsoft, *Photodna*, `https://www.microsoft.com/en-us/photodna`, (Accessed on 19/04/2022), Apr. 2022.

[6] C. Mu, J. Zhao, G. Yang, B. Yang and Z. Yan, 'Empowering elasticsearch with exact and fast r-neighbor search in hamming space,' *CoRR*, vol. abs/1902.08498, 2019. arXiv: `1902.08498`. [Online]. Available: `http://arxiv.org/abs/1902.08498`.

[7] P. B. Meggs, *A History of Graphic Design*. John Wiley Sons, Inc, 1998.

[8] A. Tirkel, G. Rankin, R. Schyndel, W. Ho, N. Mee and C. Osborne, 'Electronic water mark,' *DICTA–93*, pp. 666–672, Dec. 1993.

[9] J. Law-To, L. Chen, A. Joly, I. Laptev, O. Buisson, V. Gouet-Brunet, N. Boujemaa and F. Stentiford, 'Video copy detection: A comparative study,' Jul. 2007, pp. 371–378. DOI: `10.1145/1282280.1282336`.

[10] P. Jiang, S. He, H. Yu and Y. Zhang, 'Two-stage visible watermark removal architecture based on deep learning,' eng, *IET image processing*, vol. 14, no. 15, pp. 3819–3828, 2020, ISSN: 1751-9659.

[11] C. Gao, A. Saraf, J.-B. Huang and J. Kopf, 'Flow-edge guided video completion,' in *Proc. European Conference on Computer Vision (ECCV)*, 2020.

[12] P. C. Bjelland, K. Franke and A. Årnes, 'Practical use of approximate hash based matching in digital investigations,' eng, *Digital investigation*, vol. 11, no. 1, S18–S26, 2014, ISSN: 1742-2876.

[13] A. A. Taha and S. J. Malebary, 'Hybrid classification of android malware based on fuzzy clustering and the gradient boosting machine,' eng, *Neural computing applications*, vol. 33, no. 12, pp. 6721–6732, 2020, ISSN: 0941-0643.

[14] N. D. Gharde, D. M. Thounaojam, B. Soni and S. K. Biswas, 'Robust perceptual image hashing using fuzzy color histogram,' eng, *Multimedia tools and applications*, vol. 77, no. 23, pp. 30 815–30 840, 2018, ISSN: 1380-7501.

[15] Content Blockchain, *Testing different image hash functions*, `https://content-blockchain.org/research/testing-different-image-hash-functions/`, (Accessed on 05/04/2022), May 2022.

[16] M. C. Amirani, M. Toorani and A. Beheshti, 'A new approach to content-based file type detection,' in *2008 IEEE Symposium on Computers and Communications*, IEEE, Jul. 2008. DOI: `10.1109/iscc.2008.4625611`. [Online]. Available: `https://doi.org/10.1109%5C%2Fiscc.2008.4625611`.

[17] E. Barfian, B. H. Iswanto and S. M. Isa, 'Twitter pornography multilingual content identification based on machine learning,' eng, vol. 116, pp. 129–136, 2017, ISSN: 1877-0509.

[18] C. Nicholson, L. Beattie, M. Beattie, T. Razzaghi and S. Chen, 'A machine learning and clustering-based approach for county-level covid-19 analysis,' eng, *PloS one*, vol. 17, no. 4, e0267558–e0267558, 2022, ISSN: 1932-6203.

[19] J. Sester, D. Hayes, M. Scanlon and N.-A. Le-Khac, 'A comparative study of support vector machine and neural networks for file type identification using n-gram analysis,' *Forensic Science International Digital Investigation*, vol. 36, Mar. 2021. DOI: `10.1016/j.fsidi.2021.301121`.

[20] IBM Cloud Education, *What is deep learning?* `https://www.ibm.com/cloud/learn/deep-learning#:~:text=Deep%20learning%20is%20a%20subset,from%20large%20amounts%20of%20data.`, (Accessed on 19/05/2022), May 2020.

[21] J. Kobielus, *Gpus continue to dominate the ai accelerator market for now*, `https://www.informationweek.com/ai-or-machine-learning/gpus-continue-to-dominate-the-ai-accelerator-market-for-now`, (Accessed on 19/05/2022), Nov. 2019.

[22] 'State intellectual property office of china releases univ jiliang china's patent application for face recognition based on depth neural network multilayer feature fusion,' eng, *Global IP News. Information Technology Patent News*, 2020.

[23] W. Song, C. Chen, Q. Zhao and F. Liu, 'Spatial-temporal representation for video reidentification via key images,' eng, *IET computer vision*, vol. 14, no. 6, pp. 399–406, 2020, ISSN: 1751-9632.

[24] J. Q. Candela, *Building scalable systems to understand content - engineering at meta*, https://engineering.fb.com/2017/02/02/ml-applications/building-scalable-systems-to-understand-content/, (Accessed on 19/05/2022), Feb. 2017.

[25] K. Johnson, *The efforts to make text-based ai less racist and terrible*, https://www.wired.com/story/efforts-make-text-ai-less-racist-terrible/, (Accessed on 19/05/2022), Jun. 2021.

[26] Youtube creators, *Youtube content id*, https://www.youtube.com/watch?v=9g2U12SsRns, (Accessed on 22/05/2021), Sep. 2010.

[27] Griffeye Analyze Platform, *Using griffeye technology to increase efficiency and results*, https://www.youtube.com/watch?v=9C7HrUd-Uto?t=708, (Accessed on 18/05/2022), Aug. 2018.

[28] Griffeye Technologies, *Griffeye analyze di pro*, https://www.griffeye.com/analyze-di/, (Accessed on 22/05/2022), May 2022.

[29] Griffeye Technologies, *Griffeye intelligence database*, https://www.griffeye.com/griffeye-intelligence-database/, (Accessed on 22/05/2022), May 2022.

[30] J. Langston, *How photodna for video is being used to fight online child exploitation – on the issues*, https://news.microsoft.com/on-the-issues/2018/09/12/how-photodna-for-video-is-being-used-to-fight-online-child-exploitation/, (Accessed on 18/05/2022), Sep. 2018.

[31] H. Farid, 'Reining in online abuses,' English, *Technology and Innovation*, vol. 19, no. 3, pp. 593–599, 2018, Name - Dartmouth College; Copyright - Copyright National Academy of Inventors 2018; Last updated - 2021-09-11; SubjectsTermNotLitGenreText - United States–US. [Online]. Available: https://www.proquest.com/scholarly-journals/reining-online-abuses/docview/2013546711/se-2?accountid=12870.

[32] J. Meisner, *Facebook to use microsoft's photodna technology to combat child exploitation*, https://blogs.microsoft.com/on-the-issues/2011/05/19/facebook-to-use-microsofts-photodna-technology-to-combat-child-exploitation/, (Accessed on 22/05/2022), May 2011.

[33] A. Orr, *Apple now scans uploaded content for child abuse imagery (update)*, https://www.macobserver.com/analysis/apple-scans-uploaded-content/, (Accessed on 22/05/2022), Aug. 2021.

[34] Microsoft, *Photodna cloud service*, https://www.microsoft.com/en-us/PhotoDNA/CloudService, (Accessed on 22/05/2022), May 2022.

[35] J. Kaiser, *Cli java wrapper for the photodna library*, https://github.com/jankais3r/jPhotoDNA, (Accessed on 22/05/2022), Aug. 2021.

[36] N. Krawetz, *Photodna and limitations - the hacker factor blog*, https://hackerfactor.com/blog/index.php?/archives/931-PhotoDNA-and-Limitations.html, (Accessed on 22/05/2022), Aug. 2021.

[37] A. Athalye, *Inverting photodna*, https://www.anishathalye.com/2021/12/20/inverting-photodna/, (Accessed on 22/05/2022), Dec. 2021.

[38] Youtube, *How content id works - youtube help*, https://support.google.com/youtube/answer/2797370?hl=en, (Accessed on 22/05/2022), May 2022.

[39] J. E. Gray and N. P. Suzor, 'Playing with machines: Using machine learning to understand automated copyright enforcement at scale,' *Big Data & Society*, vol. 7, no. 1, p. 2 053 951 720 919 963, 2020. DOI: 10.1177/2053951720919963. eprint: https://doi.org/10.1177/2053951720919963. [Online]. Available: https://doi.org/10.1177/2053951720919963.

[40] P. Saadatpanah, A. Shafahi and T. Goldstein, 'Adversarial attacks on copyright detection systems,' in *International Conference on Machine Learning*, PMLR, 2020, pp. 8307–8315.

[41] K. Gardåsen, *Clipped: A solution for finding the source footage from a video clip*, https://web.archive.org/web/20180611001102/https://github.com/Data-Kjetil/CLIPPED, Dec. 2013.

[42] *Scientific method*. DOI: 10.1093/oi/authority.20110803100447727. [Online]. Available: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803100447727.

[43] Elastic, *Hardware | elasticsearch: The definitive guide [2.x]*, https://www.elastic.co/guide/en/elasticsearch/guide/current/hardware.html, (Accessed on 29/04/2022), Apr. 2022.

[44] Elastic, *Hardware prerequisites | elastic cloud enterprise reference [3.1]*, https://www.elastic.co/guide/en/cloud-enterprise/current/ece-hardware-prereq.html, (Accessed on 28/04/2022), Apr. 2022.

[45] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth and L.-J. Li, 'Yfcc100m: The new data in multimedia research,' *Communications of the ACM*, vol. 59, no. 2, pp. 64–73, 2016.

[46] Amazon, *Aws command line interface*, https://aws.amazon.com/cli/, (Accessed on 04/05/2022), May 2022.

[47] Gyan, *How to capture first x frames every x seconds into a png with ffmpeg?* https://superuser.com/questions/1388870/how-to-capture-first-x-frames-every-x-seconds-into-a-png-with-ffmpeg/1389002, (Accessed on 05/05/2022), Dec. 2018.

[48] mayya-sharipova, *Issue - forbid negative values for "weight" in function score query*, https://github.com/elastic/elasticsearch/issues/31927, (Accessed on 09/05/2022), Jul. 2018.

[49] F. W. T. Reite, *Pyclipnsearchit: A scalable approach to video indexing and search for use in the identification of fragmented and transformed image/video files*, https://github.com/WilhelmThon/pyClipNSearchIt, Jun. 2022.

[50] R. Feynman, *Feynman on scientific method*, `https://www.youtube.com/watch?v=EYPapE-3FRw`, (Accessed on 07/05/2022), 1965.

[51] GNU Project - Free Software Foundation, *The gnu general public license v3.0*, `https://www.gnu.org/licenses/gpl-3.0.en.html`, (Accessed on 18/04/2022), Jun. 2007.

[52] Wikipedia contributors, *File format (magic number) — Wikipedia, the free encyclopedia*, (Accessed on 19/04/2022), 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=File_format&oldid=1082030016#Magic_number`.

[53] FFmpeg, *Ffmpeg git repo*, `https://git.ffmpeg.org/ffmpeg.git`, (Accessed on 19/04/2022), Apr. 2022.

Appendix A

Frontend setup script

Code listing A.1: Python script for setting up the desktop/master frontend environment

```
import sys
import os
import platform
import subprocess
import json

from src import logger
from src import util

FILE_DIR = os.path.dirname(__file__)
CONFIG = {
    "elasticsearch": {
        "hosts": [
            "https://localhost:9200"
        ],
        "http_auth": [
            "user",
            "password"
        ],
        "verify_certs": False,
        "ssl_show_warn": False
    },
    "elasticsearch_index": {
        "number_of_shards": 9,
        "number_of_replicas": 1
    },
    "ffmpeg": {
        "use_hwaccel": False,
        "capture_every_n_seconds": 1
    },
    "search": {
        "max_size_response": 15,
        "r": 5
    }
}

def install_requirements_pip():
    logger.info("...Attempting␣to␣install␣pip␣requirements")
    # These are included to ensure we don’t encounter any weird errors when
        ↪ installing the requirements
    subprocess.check_call([sys.executable, "-m", "pip", "install", "--upgrade", ’
        ↪ pip’])
    subprocess.check_call([sys.executable, "-m", "pip", "install", "--upgrade", ’
        ↪ setuptools’])
    subprocess.check_call([sys.executable, "-m", "pip", "install", "--upgrade", ’
        ↪ distlib’])
    subprocess.check_call([sys.executable, "-m", "pip", "install", "--upgrade", ’
        ↪ requests’])
    # Install requirements
    subprocess.check_call([sys.executable, "-m", "pip", "install", "-r", os.path.
        ↪ join(FILE_DIR, "requirements.txt")])
    logger.info("...Success,␣installed␣pip␣requirements")

def install_requirements_windows():
    import ssl
    from urllib.request import urlopen, Request
    from urllib.error import URLError
    from http.client import IncompleteRead
    from io import BytesIO
    from zipfile import ZipFile

    # Download and install ffmpeg
    try:
        logger.info("...Attempting␣to␣download␣ffmpeg")
        http_header = {’User-Agent’: ’Mozilla/5.0␣(Windows␣NT␣10.0;␣Win64;␣x64)␣
            ↪ AppleWebKit/537.36␣(KHTML,␣like␣Gecko)␣Chrome/70.0.3538.77␣Safari
            ↪ /537.36’}

        context = ssl._create_unverified_context()

        with urlopen(Request(
            url="https://www.gyan.dev/ffmpeg/builds/git-version",
            headers=http_header),
            context=context
        ) as response:
            content = response.read()
            encoding = response.headers.get_content_charset(’utf-8’)
            ffmpeg_version = content.decode(encoding)

        logger.info(f"...Latest␣ffmpeg␣git␣version␣is␣{ffmpeg_version}")

        with urlopen(Request(
            url=f"https://github.com/GyanD/codexffmpeg/releases/download/{
                ↪ ffmpeg_version}/ffmpeg-{ffmpeg_version}-essentials_build.zip",
            headers=http_header),
            context=context
        ) as response:
            url_file = BytesIO(response.read())
        logger.info("...Success,␣downloaded␣ffmpeg")

    except (URLError, IncompleteRead) as e:
        logger.error(f"Failed␣to␣download␣ffmpeg␣’{e}’")
        quit()

    with ZipFile(url_file, mode = ’r’) as zipObj:
        logger.info("...Extracting␣ffmpeg␣from␣archive")
        for zip_info in zipObj.infolist():
            if zip_info.filename[-1] == ’/’: continue
            basename = os.path.basename(zip_info.filename)

            if not basename == 'ffmpeg.exe': continue

            zip_info.filename = basename
            zipObj.extract(zip_info, path=FILE_DIR)
            break
        logger.info("...Success,␣extracted␣ffmpeg.exe␣from␣archive")

def install_requirements_linux():
    logger.info("...Attempting␣to␣download␣dependencies")
    subprocess.check_call(['sudo', 'add-apt-repository', 'ppa:savoury1/ffmpeg4'],
        ↪ stdout=sys.stdout, stderr=sys.stderr)
    subprocess.check_call(['sudo', 'apt', 'install'] + ['ffmpeg', 'libjpeg8-dev', '
        ↪ zlib1g-dev', 'python3.9-distutils'] + ['-y'], stdout=sys.stdout, stderr
        ↪ =sys.stderr)
    logger.info("...Success,␣dependencies␣downloaded")

def main():
    logger.info("Running␣frontend␣setup!")

    if util.is_frontend_setup():
        logger.error("setup_frontend.py␣has␣already␣been␣run!")
        return

    system_name = platform.system()

    # System specific install requirements
    if system_name == 'Windows':
        install_requirements_windows()
    elif system_name == 'Linux':
        install_requirements_linux()
    else:
        logger.error("You␣are␣not␣on␣a␣supported␣system")
        return

    install_requirements_pip()

    # Create temp data folder
    tmp_dir = os.path.join(util.ROOT_DIR, ".tmp")
    os.mkdir(tmp_dir)

    if system_name == 'Windows':
        os.system(f"attrib␣+h␣{tmp_dir}")

    # Create config
    config_file = os.path.join(util.ROOT_DIR, "config.json")
    with open(config_file, 'w+') as file:
        json.dump(CONFIG, file, indent=4)

    logger.info("Created␣config.json.␣Remember␣to␣edit␣the␣default␣configuration!")
    logger.info("Finished␣running␣frontend␣setup!")

if __name__ == "__main__":
    main()
```

Appendix B

Default pyClipNSearchIt configuration

Code listing B.1: JSON file for configuring the pyClipNSearchIt settings

```
{
    "elasticsearch": {
        "hosts": [
            "https://localhost:9200"
        ],
        "http_auth": [
            "user",
            "password"
        ],
        "verify_certs": false,
        "ssl_show_warn": false
    },
    "elasticsearch_index": {
        "number_of_shards": 9,
        "number_of_replicas": 1
    },
    "ffmpeg": {
        "use_hwaccel": false,
        "capture_every_n_seconds": 3
    },
    "search": {
        "max_size_response": 50,
        "r": 5
    }
}
```

Appendix C

Backend setup script

Code listing C.1: Python script for setting up the Elasticsearch backend environment

```python
import sys

from src import logger
from src import util

HMDSCRIPT_TEMPLATE = {
    "id": "hmd64bit",
    "script": {
        "lang": "painless",
        "source": """
            long u = params.subcode^doc[params.field].value;
            long uCount = u-((u>>>1)&-5270498306774157605L)-((u>>>2)
                ↪ &-7905747460161236407L);
            return ((uCount+(uCount>>>3))&8198552921648689607L)%63;
        """
    }
}

PIPELINE_TEMPLATE = {
    "id": "split-hash",
    "description": "Splits␣the␣input␣hashes␣into␣smaller␣subsets",
    "processors": [
        {
            "set": {
                "field": "fhash",
                "value": {}
            }
        },
        {
            "script" : {
                "lang": "painless",
                "source": """
                    ctx.bhash = new BigInteger(ctx.hash, 16).longValue();
                    ctx.fhash.f1 = ctx.hash.substring(0, 4);
                    ctx.fhash.f2 = ctx.hash.substring(4, 8);
                    ctx.fhash.f3 = ctx.hash.substring(8, 12);
                    ctx.fhash.f4 = ctx.hash.substring(12, 16);
                """
            }
        }
    ]
}

INDEX_TEMPLATE = {
    "index": "hashes",
    "settings" : {
        "number_of_shards": util.get_config()["elasticsearch_index"]["
            ↪ number_of_shards"],
        "number_of_replicas": util.get_config()["elasticsearch_index"]["
            ↪ number_of_replicas"],
        "default_pipeline": "split-hash"
    },
    "mappings": {
        "properties": {
            "origin": { "type": "text" },
            "timestamp": { "type": "integer" },
            "hash": { "type": "keyword" },
            "bhash": { "type": "long" },
            "fhash": {
                "properties": {
                    "f1": { "type": "keyword" },
                    "f2": { "type": "keyword" },
                    "f3": { "type": "keyword" },
                    "f4": { "type": "keyword" },
                }
            }
        }
    }
}

def clear(client):
    logger.info("...Attempting␣to␣delete␣old␣pipelines␣and␣indexes")
    try:
        client.indices.delete(index=INDEX_TEMPLATE["index"])
        client.ingest.delete_pipeline(id=PIPELINE_TEMPLATE["id"])
        client.delete_script(id=HMDSCRIPT_TEMPLATE["id"])
    except:
        pass
    logger.info("...Success,␣deleted␣old␣pipelines␣and␣indexes")

def setup(client):
    logger.info("...Attempting␣to␣create␣hemming␣distance␣script")
    client.put_script(**HMDSCRIPT_TEMPLATE)
    logger.info("...Success,␣created␣hemming␣distance␣script")

    logger.info("...Attempting␣to␣create␣split-hash␣pipeline")
    client.ingest.put_pipeline(**PIPELINE_TEMPLATE)
    logger.info("...Success,␣created␣split-hash␣pipeline")

    logger.info("...Attempting␣to␣create␣hash␣index")
    client.indices.create(**INDEX_TEMPLATE)
    logger.info("...Success,␣created␣hash␣index")

def main():
    logger.info("Running␣backend␣setup!")

    if not util.is_frontend_setup():
        logger.error("setup_frontend.py␣must␣be␣run␣first")
        return
    # Import later to not give error if frontend has not been ran
    import src.database as database

    try:
        client = database.connect()
    except Exception as e:
        logger.error(str(e))

    if len(sys.argv) > 1 and sys.argv[1] == "--force":
        logger.info("Forcing␣clean␣setup")
        clear(client)

    if util.is_backend_setup(client):
        logger.error("The␣database␣is␣already␣setup")
        return

    setup(client)

    logger.info("Finished␣running␣backend␣setup!")

if __name__ == "__main__":
    main()
```

Appendix D

Elasticsearch nodes configuration

Code listing D.1: Elasticsearch.yml node-1 (master) configuration

```
# ======================== Elasticsearch Configuration =========================
#
# NOTE: Elasticsearch comes with reasonable defaults for most settings.
#       Before you set out to tweak and tune the configuration, make sure you
#       understand what are you trying to accomplish and the consequences.
#
# The primary way of configuring a node is via this file. This template lists
# the most important settings you may want to configure for a production cluster.
#
# Please consult the documentation for further information on configuration options
    ↪ :
# https://www.elastic.co/guide/en/elasticsearch/reference/index.html
#
# ---------------------------------- Cluster -----------------------------------
#
# Use a descriptive name for your cluster:
#
cluster.name: hash-cluster
#
# ------------------------------------ Node ------------------------------------
#
# Use a descriptive name for the node:
#
node.name: node-1
node.roles: [ master, data]
#
# Add custom attributes to the node:
#
#node.attr.rack: r1
#
# ----------------------------------- Paths ------------------------------------
#
# Path to directory where to store the data (separate multiple locations by comma):
# This path is a mounted 1TB HDD. Subsituting it for a SSD would likely yield much
    ↪ better indexing/search  speeds
path.data: /data/elastic
#
```

```
# Path to log files:
#
path.logs: /var/log/elasticsearch
#
# ----------------------------------- Memory -----------------------------------
#
# Lock the memory on startup:
#
#bootstrap.memory_lock: true
#
# Make sure that the heap size is set to about half the memory available
# on the system and that the owner of the process is allowed to use this
# limit.
#
# Elasticsearch performs poorly when the system is swapping the memory.
#
# ---------------------------------- Network -----------------------------------
#
# By default Elasticsearch is only accessible on localhost. Set a different
# address here to expose this node on the network:
#
network.host: 192.168.0.35
#
# By default Elasticsearch listens for HTTP traffic on the first free port it
# finds starting at 9200. Set a specific HTTP port here:
#
#http.port: 9200
#
# For more information, consult the network module documentation.
#
# --------------------------------- Discovery ----------------------------------
#
# Pass an initial list of hosts to perform discovery when this node is started:
# The default list of hosts is ["127.0.0.1", "[::1]"]
#
#discovery.seed_hosts: ["host1", "host2"]
#
# Bootstrap the cluster using an initial set of master-eligible nodes:
#
#cluster.initial_master_nodes: ["node-1", "node-2"]
#
# For more information, consult the discovery and cluster formation module
    ↪ documentation.
#
# ---------------------------------- Various -----------------------------------
#
# Allow wildcard deletion of indices:
#
#action.destructive_requires_name: false

#----------------------- BEGIN SECURITY AUTO CONFIGURATION -----------------------
#
# The following settings, TLS certificates, and keys have been automatically
# generated to configure Elasticsearch security features on 14-03-2022 09:40:26
#
# --------------------------------------------------------------------------------

# Enable security features
xpack.security.enabled: true
xpack.security.enrollment.enabled: true

# Enable encryption for HTTP API client connections, such as Kibana, Logstash, and
    ↪ Agents
xpack.security.http.ssl:
  enabled: true
  keystore.path: certs/http.p12

# Enable encryption and mutual authentication between cluster nodes
xpack.security.transport.ssl:
  enabled: true
  verification_mode: certificate
  keystore.path: certs/transport.p12
  truststore.path: certs/transport.p12
# Create a new cluster with the current node only
# Additional nodes can still join the cluster later
cluster.initial_master_nodes: ["node-1"]

# Allow HTTP API connections from localhost and local networks
# Connections are encrypted and require user authentication
http.host: [_local_, _site_]

# Allow other nodes to join the cluster from localhost and local networks
# Connections are encrypted and mutually authenticated
#transport.host: [_local_, _site_]

#----------------------- END SECURITY AUTO CONFIGURATION -------------------------
```

Code listing D.2: Elasticsearch.yml node-2 (data, ingest) configuration

```
# ======================== Elasticsearch Configuration =========================
#
# NOTE: Elasticsearch comes with reasonable defaults for most settings.
#       Before you set out to tweak and tune the configuration, make sure you
#       understand what are you trying to accomplish and the consequences.
#
# The primary way of configuring a node is via this file. This template lists
# the most important settings you may want to configure for a production cluster.
#
# Please consult the documentation for further information on configuration options
    ↪ :
# https://www.elastic.co/guide/en/elasticsearch/reference/index.html
#
# ---------------------------------- Cluster -----------------------------------
#
# Use a descriptive name for your cluster:
#
cluster.name: hash-cluster
#
# ------------------------------------ Node ------------------------------------
#
# Use a descriptive name for the node:
#
node.name: node-2
node.roles: [ data, ingest ]
#
# Add custom attributes to the node:
#
#node.attr.rack: r1
#
```

```
# ---------------------------------- Paths -----------------------------------
#
# Path to directory where to store the data (separate multiple locations by comma):
# This path is a mounted 1TB HDD. Subsituting it for a SSD would likely yield much
    ↪ better indexing/search  speeds
path.data: /data/elastic
#
# Path to log files:
#
path.logs: /var/log/elasticsearch
#
# ---------------------------------- Memory -----------------------------------
#
# Lock the memory on startup:
#
#bootstrap.memory_lock: true
#
# Make sure that the heap size is set to about half the memory available
# on the system and that the owner of the process is allowed to use this
# limit.
#
# Elasticsearch performs poorly when the system is swapping the memory.
#
# --------------------------------- Network -----------------------------------
#
# By default Elasticsearch is only accessible on localhost. Set a different
# address here to expose this node on the network:
#
network.host: 192.168.0.149
#
# By default Elasticsearch listens for HTTP traffic on the first free port it
# finds starting at 9200. Set a specific HTTP port here:
#
#http.port: 9200
#
# For more information, consult the network module documentation.
#
# --------------------------------- Discovery ----------------------------------
#
# Pass an initial list of hosts to perform discovery when this node is started:
# The default list of hosts is ["127.0.0.1", "[::1]"]
#
#discovery.seed_hosts: ["192.168.0.35:9300"]
#
# Bootstrap the cluster using an initial set of master-eligible nodes:
#
#cluster.initial_master_nodes: ["node-1"]
#
# For more information, consult the discovery and cluster formation module
    ↪ documentation.
#
# --------------------------------- Various -----------------------------------
#
# Allow wildcard deletion of indices:
#
#action.destructive_requires_name: false

#----------------------- BEGIN SECURITY AUTO CONFIGURATION -----------------------
#
# The following settings, TLS certificates, and keys have been automatically
# generated to configure Elasticsearch security features on 14-03-2022 09:44:44
#
# --------------------------------------------------------------------------------

# Enable security features
xpack.security.enabled: true

xpack.security.enrollment.enabled: true

# Enable encryption for HTTP API client connections, such as Kibana, Logstash, and
    ↪ Agents
xpack.security.http.ssl:
  enabled: true
  keystore.path: certs/http.p12

# Enable encryption and mutual authentication between cluster nodes
xpack.security.transport.ssl:
  enabled: true
  verification_mode: certificate
  keystore.path: certs/transport.p12
  truststore.path: certs/transport.p12
# Discover existing nodes in the cluster
discovery.seed_hosts: ["192.168.0.35"]

# Allow HTTP API connections from localhost and local networks
# Connections are encrypted and require user authentication
http.host: [_local_, _site_]

# Allow other nodes to join the cluster from localhost and local networks
# Connections are encrypted and mutually authenticated
#transport.host: [_local_, _site_]

#----------------------- END SECURITY AUTO CONFIGURATION -------------------------
```

Code listing D.3: Elasticsearch.yml node-3 (data, ingest) configuration

```
# ======================== Elasticsearch Configuration =========================
#
# NOTE: Elasticsearch comes with reasonable defaults for most settings.
#       Before you set out to tweak and tune the configuration, make sure you
#       understand what are you trying to accomplish and the consequences.
#
# The primary way of configuring a node is via this file. This template lists
# the most important settings you may want to configure for a production cluster.
#
# Please consult the documentation for further information on configuration options
    ↪ :
# https://www.elastic.co/guide/en/elasticsearch/reference/index.html
#
# ---------------------------------- Cluster -----------------------------------
#
# Use a descriptive name for your cluster:
#
cluster.name: hash-cluster
#
# ------------------------------------ Node ------------------------------------
#
# Use a descriptive name for the node:
#
node.name: node-3
node.roles: [ data, ingest ]
#
# Add custom attributes to the node:
#
#node.attr.rack: r1
#
# ---------------------------------- Paths -----------------------------------
#
# Path to directory where to store the data (separate multiple locations by comma):
# This path is a mounted 1TB HDD. Subsituting it for a SSD would likely yield much
    ↪ better indexing/search  speeds
path.data: /data/elastic
#
# Path to log files:
#
path.logs: /var/log/elasticsearch
#
# ---------------------------------- Memory ----------------------------------
#
# Lock the memory on startup:
#
#bootstrap.memory_lock: true
#
# Make sure that the heap size is set to about half the memory available
# on the system and that the owner of the process is allowed to use this
# limit.
#
# Elasticsearch performs poorly when the system is swapping the memory.
#
# --------------------------------- Network ----------------------------------
#
# By default Elasticsearch is only accessible on localhost. Set a different
# address here to expose this node on the network:
#
network.host: 192.168.0.61
#
# By default Elasticsearch listens for HTTP traffic on the first free port it
# finds starting at 9200. Set a specific HTTP port here:
#
#http.port: 9200
#
# For more information, consult the network module documentation.
#
# --------------------------------- Discovery --------------------------------
#
# Pass an initial list of hosts to perform discovery when this node is started:
# The default list of hosts is ["127.0.0.1", "[::1]"]
#
#discovery.seed_hosts: ["host1", "host2"]
#
# Bootstrap the cluster using an initial set of master-eligible nodes:
#
#cluster.initial_master_nodes: ["node-1", "node-2"]
#
# For more information, consult the discovery and cluster formation module
    ↪ documentation.
#
# --------------------------------- Various ----------------------------------
#
# Allow wildcard deletion of indices:
```

```
#
#action.destructive_requires_name: false

#----------------------- BEGIN SECURITY AUTO CONFIGURATION -----------------------
#
# The following settings, TLS certificates, and keys have been automatically
# generated to configure Elasticsearch security features on 14-03-2022 09:48:10
#
# --------------------------------------------------------------------------------

# Enable security features
xpack.security.enabled: true

xpack.security.enrollment.enabled: true

# Enable encryption for HTTP API client connections, such as Kibana, Logstash, and
    ↪ Agents
xpack.security.http.ssl:
  enabled: true
  keystore.path: certs/http.p12

# Enable encryption and mutual authentication between cluster nodes
xpack.security.transport.ssl:
  enabled: true
  verification_mode: certificate
  keystore.path: certs/transport.p12
  truststore.path: certs/transport.p12
# Discover existing nodes in the cluster
discovery.seed_hosts: ["192.168.0.35"]

# Allow HTTP API connections from localhost and local networks
# Connections are encrypted and require user authentication
http.host: [_local_, _site_]

# Allow other nodes to join the cluster from localhost and local networks
# Connections are encrypted and mutually authenticated
#transport.host: [_local_, _site_]

#----------------------- END SECURITY AUTO CONFIGURATION -------------------------
```

Appendix E

Multimedia Commons download script

Code listing E.1: Python script for downloading a small subset of the Multimedia Commons dataset

```python
import sys
import os
import subprocess

from src import logger

def main():
    if len(sys.argv) != 2:
        logger.error("Requires␣an␣output␣directory")
        return

    output = os.path.abspath(sys.argv[1])

    if not os.path.isdir(output):
        logger.error("Ouput␣path␣is␣not␣a␣directory")
        return

    base_cmd_images = f"python␣-m␣awscli␣s3␣sync␣--no-sign-request␣s3://multimedia-
        ↪ commons/data/images/{{0}}␣{output}/images/{{0}}"
    base_cmd_videos = f"python␣-m␣awscli␣s3␣sync␣--no-sign-request␣s3://multimedia-
        ↪ commons/data/videos/mp4/{{0}}␣{output}/videos/{{0}}"

    logger.info(f"Downloading␣the␣dataset")

    # Yes I know these loops are basically identical.
    # Yes they can be improved.
    for i in range(0, 256, 8):
        p_handles = []

        try:
            for j in range (i, i + 8):
                hex_i = hex(j)[2:].rjust(3, '0')
                p_handles.append(subprocess.Popen(base_cmd_images.format(hex_i),
                    ↪ stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL))
                p_handles.append(subprocess.Popen(base_cmd_videos.format(hex_i),
                    ↪ stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL))
            for p_handle in p_handles:
                p_handle.wait()
        except KeyboardInterrupt as e:
            logger.error("Interrupted␣download.␣Terminating␣running␣download␣
                ↪ processes")

            for p_handle in p_handles:
                p_handle.terminate()

            raise e

        logger.info(f"Downloaded␣{i␣+␣8}/256␣of␣the␣general␣dataset")

    for i in range(256, 512, 8):
        p_handles = []

        try:
            for j in range (i, i + 8):
                hex_i = hex(j)[2:].rjust(3, '0')
                p_handles.append(subprocess.Popen(base_cmd_videos.format(hex_i),
                    ↪ stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL))

            for p_handle in p_handles:
                p_handle.wait()
        except KeyboardInterrupt as e:
            logger.error("Interrupted␣download.␣Terminating␣running␣download␣
                ↪ processes")

            for p_handle in p_handles:
                p_handle.terminate()

            raise e

        logger.info(f"Downloaded␣{(i␣-␣248)␣}/256␣of␣the␣additional␣video␣dataset")

    logger.info(f"Finished␣downloading␣the␣dataset")

if __name__ == "__main__":
    main()
```

Appendix F

Tool - Count hashes script

Code listing F.1: Python script for counting the number of hashes in a .pregenhashes file

```
import sys
import os
import json

sys.path.insert(1, os.path.join(sys.path[0], '..'))

from src import logger
from src import util
from src.structures import FileType

def main():
    logger.info("Running␣count␣hashes!")

    if not util.is_frontend_setup():
        logger.error("setup_frontend.py␣must␣be␣run␣first")
        return

    if len(sys.argv) != 2:
        logger.error("You␣must␣provide␣a␣.pregenhashes␣or␣.pregenhashes.
            ↪ notnormalized.txt␣input␣file")
        return

    path = os.path.abspath(sys.argv[1])

    if (not os.path.isfile(path)) or (util.get_file_type(path) != FileType.HASHES
         ↪ and not path.lower().endswith(".pregenhashes.notnormalized.txt")):
        logger.error("The␣provided␣path␣is␣either␣not␣valid␣or␣the␣correct␣file␣
            ↪ type␣(.pregenhashes,␣.pregenhashes.notnormalized.txt)")
        return

    with open(path) as file:
        hash_sets = json.load(file)

    number_of_hashes = 0

    for origin in hash_sets:
        data = hash_sets[origin]

        if type(data) is str:
            number_of_hashes += 1
        else:
            number_of_hashes += len(data)

    logger.info(f"The␣number␣of␣hashes␣in␣{os.path.basename(path)}␣is␣{
        ↪ number_of_hashes}")
    logger.info("Finished␣running␣count␣hashes!")

if __name__ == "__main__":
    main()
```

Appendix G

Tool - Extract clips script

Code listing G.1: Python script for extracting fragments/clips from videos

```python
import sys
import os
import subprocess
import imagehash

sys.path.insert(1, os.path.join(sys.path[0], '..'))

from collections import defaultdict
from PIL import Image
from distutils.util import strtobool
from src import logger
from src import util
from src.arguments import _is_valid_input, _is_valid_output
from src.hasher import preprocess, _write_to_file

def hash_videos(videos: list):
    if not videos:
        return None

    tmp_dir = os.path.join(util.ROOT_DIR, ".tmp")
    base_cmd = util.get_converter(ignore_settings=True)
    videos_len = len(videos)
    hash_sets = defaultdict(list)

    for i in range(0, videos_len, 20):
        videos_left = videos_len - i
        remaining = videos_left if videos_left < 20 else 20

        for j in range(i, i + remaining):
            output = os.path.join(tmp_dir, f"{os.path.basename(videos[j])}.%d")
            cmd = base_cmd.format(videos[j], output)

            logger.info(cmd)

            subprocess.Popen(cmd, shell=True).wait()

        files = []

        with os.scandir(tmp_dir) as it:
            for entry in it:
                files.append(entry.path)

        for file in files:
            try:
                image = Image.open(file)
                name, frame = os.path.splitext(os.path.basename(file))
                hash_sets[name].append((str(imagehash.phash(image, 8)), int(frame
                    ↪ [1:]) - 1))
            except:
                pass

            hash_sets[name].sort(key=lambda x: x[1])

            os.remove(file)

    return hash_sets

def extract_clips(hash_sets: dict, length, offset):
    to_remove = []

    for origin in hash_sets:
        data = hash_sets[origin]
        video_length = len(data)

        if video_length < (offset + length):
            logger.info(f"The␣chosen␣offset␣'{offset}'␣and␣length␣'{length}'␣=␣{
                ↪ offset+length},␣exceeds␣␣the␣video␣length␣of␣{origin},␣'{
                ↪ video_length}'.␣Skipping␣the␣clip")
            to_remove.append(origin)
            continue

        hash_sets[origin] = data[offset:offset+length]

    for origin in to_remove:
        del hash_sets[origin]

def normalize_output(hash_sets: dict):
    frame_capture_setting = util.get_config()["ffmpeg"]["capture_every_n_seconds"]

    for origin in hash_sets:
        data = hash_sets[origin]
        tmp_data = {}

        for i in range(0, len(data), frame_capture_setting):
            frame, timestamp = data[i]
            tmp_data[frame] = timestamp

        hash_sets[origin] = tmp_data

def main():
    logger.info("Running␣extract␣clips!")

    if not util.is_frontend_setup():
        logger.error("setup_frontend.py␣must␣be␣run␣first")
        return

    num_args = len(sys.argv)

    if num_args < 4 or num_args > 6:
        logger.error("You␣must␣provide␣a␣valid␣input␣path,␣output␣path,␣clip␣length
            ↪ ␣(int),␣and␣optionally␣an␣offset␣(int,␣default␣0)␣and␣normalize␣
            ↪ option␣(bool,␣default␣true)")
        return

    try:
        input_path = _is_valid_input(sys.argv[1])
        output_path = _is_valid_output(sys.argv[2])
        length = int(sys.argv[3])
        offset = int(sys.argv[4]) if len(sys.argv) > 4 else 0
        normalize = strtobool(sys.argv[5].lower()) if len(sys.argv) > 5 else True

        if length <= 0:
            raise Exception("The␣length␣must␣be␣above␣0")

        _, videos, _ = preprocess.run(input_path)
        hash_sets = hash_videos(videos)

        if not hash_sets:
            raise Exception("No␣hashes␣could␣be␣calculated␣or␣read␣from␣the␣input")
    except Exception as e:
        logger.error(str(e))
        return

    # Extract the clips for each video
    extract_clips(hash_sets, length, offset)

    if normalize:
        # Normalize the output so the .pregenhashes can be read
        normalize_output(hash_sets)
    else:
        output_path += ".notnormalized.txt"


    _write_to_file(output_path, hash_sets)

    logger.info("Finished␣running␣extract␣clips!")

if __name__ == "__main__":
    main()
```

Appendix H

Tool - Transform videos script

Code listing H.1: Python script for applying transformations to a video set

```python
import sys
import os
import platform
import subprocess

sys.path.insert(1, os.path.join(sys.path[0], ’..’))

from src import logger
from src import util
from src.arguments import _is_valid_input
from src.hasher import preprocess

def get_valid_output(path: str):
    abs_path = os.path.abspath(path)

    if os.path.isfile(abs_path):
        raise TypeError("The␣output␣path␣cannot␣be␣a␣file")

    os.makedirs(abs_path, exist_ok=True)

    return abs_path

def get_converters():
    system_name = platform.system()

    if system_name == ’Windows’:
        converter = os.path.join(util.ROOT_DIR, "ffmpeg.exe")
    elif system_name == ’Linux’:
        converter = "ffmpeg"

    converters = []

    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,eq=brightness
        ↪ =0.25\"␣\"{{}}\"", "brightness_high"))
    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,eq=brightness
        ↪ =-0.25\"␣-pix_fmt␣yuv420p␣\"{{}}\"", "brightnesss_low"))
    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,eq=contrast
        ↪ =0.25\"␣-pix_fmt␣yuv420p␣\"{{}}\"", "contrast_high"))
    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,eq=contrast
        ↪ =-0.25\"␣-pix_fmt␣yuv420p␣\"{{}}\"", "contrast_low"))
    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,rotate=10*PI
        ↪ /180\"␣-pix_fmt␣yuv420p␣\"{{}}\"", "rotate"))
    converters.append((f"{converter}␣-nostdin␣-y␣-v␣0␣-vsync␣passthrough␣-i␣
        ↪ \"{{}}\"␣-vf␣\"pad=width=ceil(iw/2)*2:height=ceil(ih/2)*2,crop=0.75*
        ↪ in_w:0.75*in_h\"␣-pix_fmt␣yuv420p␣\"{{}}\"", "crop"))

    return converters

def transform_videos(videos: list, path: str):
    converters = get_converters()

    for video in videos:
        for converter_cmd, convert_type in converters:
            output = os.path.join(path, f"{convert_type}_{os.path.basename(video)}"
                ↪ )
            cmd = converter_cmd.format(video, output)

            logger.info(cmd)

            subprocess.Popen(cmd, shell=True).wait()

def main():
    logger.info("Running␣transform␣videos!")

    if not util.is_frontend_setup():
        logger.error("setup_frontend.py␣must␣be␣run␣first")
        return

    if len(sys.argv) != 3:
        logger.error("You␣must␣provide␣a␣valid␣input␣path␣and␣output␣directory")
        return

    try:
        input_path = _is_valid_input(sys.argv[1])

        _, videos, _ = preprocess.run(input_path)

        if not videos:
            raise Exception("No␣videos␣were␣found␣at␣the␣input␣path")

        output_path = get_valid_output(sys.argv[2])
    except Exception as e:
        logger.error(str(e))
        return

    transform_videos(videos, output_path)

    logger.info("Finished␣transform␣videos!")

if __name__ == "__main__":
    main()
```

Fredrik Wilhelm Thon Reite

A scalable approach to video indexing and search