Challenges with Per-Frame Metadata
With Focus on Scalability

by

Thomas Sundberg

A thesis presented for the degree of
Master of Science in Computing Science and Engineering
300 credits

Supervisor: Lars-Erik Janlert
Examinator: Henrik Björklund
June 23, 2016
Abstract

Video metadata opens up a lot of possibilities both for users and for content managers. Recently tools for automatically creating metadata via speech-to-text, face recognition and object tracking among other techniques have made metadata even more relevant. Metadata for object tracking also creates the problem of needing to be bound to individual metadata frames. This thesis tries to find what the challenges are with per-frame metadata and how it could be stored in a way that scales vertically. A pre-study was made which tried to discover possible ways to deal with it and find pros and cons with each alternative. PostgreSQL (a relational dbms) was deemed the best alternative and the performance of it was tested by running a series of queries with different population levels. The results were that the search times seemed to be log(n) which means that it scales well. The relational database proved to work well in the other aspects as well.
ACKNOWLEDGEMENTS

I would like to thank my instructor, professor Lars-Erik Janlert for providing support and always being quick to answer, reading through the thesis and always having a positive outlook.

I would also like to express my gratitude to CodeMill for the work place and the friendly attitude, I have felt very welcome during my work with this thesis. Many thanks especially to my external instructor Emil Lundh who have always seemed to have time for me when needed.

Furthermore I would like to thank all my friends and family for their support and ability to calm me down when stressed.

Finally I would like to express my deepest gratitude to my fiancée Frida, without whom I doubt I would have managed to complete this thesis. Thank you for always listening, always supporting and always being there.
Contents

1 INTRODUCTION .......................... 1
  1.1 Background .......................... 1
  1.2 The Project ......................... 2
  1.3 Goals ............................... 3

2 METHOD ............................ 4

3 PRE-STUDY ........................... 5
  3.1 Multimedia file formats ............. 5
  3.2 Annodex ........................... 6
  3.3 NoSQL Database ..................... 6
    3.3.1 Pros ........................... 6
    3.3.2 Cons ........................... 7
  3.4 Relational Database ................. 7
    3.4.1 Pros ........................... 7
    3.4.2 Cons ........................... 7
  3.5 Conclusion ......................... 8

4 TECHNICAL DETAILS ................. 9
  4.1 Choice of DBMS ..................... 9
  4.2 Database Schema .................... 10
  4.3 Queries ........................... 12
    4.3.1 Creating Tables ................. 12
    4.3.2 Insertion ....................... 13
    4.3.3 Selection ....................... 13

5 TESTS ............................. 14
  5.1 Test Machine ....................... 14
  5.2 DBMS-Configuration ................. 14
  5.3 Filling The Database ............... 14
  5.4 Vertical Scaling Tests ............. 15
Chapter 1

INTRODUCTION

1.1 Background

In today’s internet climate where youtube and netflix alone account for more than half of all internet traffic [1], there’s an increasing need to be able to store more high quality metadata for videos in a searchable way.

Smart solutions that can extract information about a video by means of voice-detection, speech-recognition, object-tracking and other intelligent algorithms can thus be extremely useful in many ways if the gathered data can be searched. For example these can be used to automatically make digitalized archives searchable without having to manually set the metadata for each video, which would be an extremely large task. Another application area of these techniques is that it could make for smarter video watching, one example is to automatically track and make it possible to click products in videos leading the user to the product’s website. These techniques could also be used for web shops to make videos where clicking an item in a video adds it to the user’s shopping cart.

This makes video analysis combined with video storage a very interesting topic with near unlimited application areas.

The company Codemill with which this thesis was carried out are developing a platform for smart video which can do the aforementioned video analysis, are looking for ways to store the metadata in a way that would scale vertically well enough to be usable for large data-sets.
1.2 The Project

This thesis will focus on how to store three distinct types of metadata in a way that makes searching the metadata possible:

**Transcript metadata:** What text is being shown in the video and at what times.

**Global metadata:** Information about the whole of the video; could be for example if the video is amateur made or professionally, length of the video, what camera was used to film it etc.

**Face (Object) tracking metadata:** Information surrounding objects in the video. This needs to be able to contain information about when the object is shown, where in the video it is placed and what object it is.

The challenge is that the face tracking and transcript metadata is gathered at a per frame level, meaning that each frame of a video needs to have corresponding metadata about what is happening in that individual frame. The main focus for the thesis will be on the face tracking metadata since the transcript metadata should be able to be stored in a similar way but without the need for position data, and the global metadata should be the least challenging to store.

The idea is that the result of this thesis can be used for a wide array of platforms that need per-frame metadata even though the report will focus on face tracking. This will be done by using the scenario of a platform that automatically finds faces and tracks these throughout a video. The resulting intervals when each face is shown and when multiple faces overlap in a video should then be searchable on a website. Users of the website should then be able to identify any unknown faces leading to that face being identified and searchable for all intervals in all videos that contain the same face. An example of the kind of XML data the face recognition program for this scenario can generate can be found in appendix A.

An example of what a user could do on the platform in this scenario is search using a string such as "Foo Barson" and receive a list of intervals that could be "Video A @ 0:20-0:30", Video B @ 0:00-3:00 & 3:30-4:00" or a search like "Foo Barson & Bar Fooson" and receive a similar list containing all intervals in which both appear.

Another part of the scenario that needs to be covered is the storage of
faces. If a new video is run through the face-recognition program and a face is found, that face should be compared against previously known faces from other videos and if a match is found it should be registered as the same face. If it is not found it should be added to the list of faces.

A mockup of a potential smart video platform where a user has searched on a video title and can see the matching videos along with the people appearing in it can be seen in 1.1.

1.3 Goals

The main goal is to gather knowledge about how video metadata and specifically per-frame metadata can be stored, and the difficulties that must be considered when designing a system like this. Another goal is to find a possible way to do this that scales vertically. Our requirement for something to be considered scalable is that the search times compared to the data sets (verticality) should increase by $O(\log n)$. The requirement for disk usage is that it should not be much worse than a linear relationship between the amount of rows in the database and the disk size.
Chapter 2

METHOD

The project began with a pre-study. The purpose of the pre-study was to find out different methods that could be used to create a searchable system for per-frame metadata that would satisfy our goals. When the different methods had been found and their strengths and weaknesses been listed one of them were chosen.

The next step was to decide on some technical decisions for the chosen method, such as creating database schemas and figuring out in more detail how the system would be designed.

Then a simplified technical demo of the proposed solution were implemented and tested for the ability to add videos & metadata (less important) and the ability to search and find videos & metadata (more important), the amount of disk space used by the database was also measured.

The scalability tests for searching data was done by running a series of queries of varying complexity at different population levels of the database i.e. by running a series of tests, then adding more videos to the database and repeat.
Chapter 3

PRE-STUDY

The pre-study began by searching Google scholar for papers about metadata and more specifically about video metadata connected to individual frames. Most papers about per-frame metadata that were found were discussing different video formats that supported per-frame metadata. These formats are however not made for searching in. Instead they are made for having metadata that can be shown or handled otherwise during playback. This makes them unsuitable for this project.

3.1 Multimedia file formats

There are various file format that can contain and make use of video metadata such as MPEG-7 and MPEG-21. There are however various immediate drawbacks with such an approach that means that we won’t investigate it further. First of all it means that such a file format will have to be supported by all media players that will be used on the platforms that will make use of my solution. This limits us to standardized formats that have widespread use. We would also like to be able to store videos of different file formats in this database.

Another drawback is that it very much limits our searching capabilities and gives very little room for customization to perfectly suit our needs. Furthermore it seems highly unlikely that searching the metadata of these formats will be as quick as other potential solutions.
3.2 Annodex

One paper was found on a general method for handling video metadata bound to time intervals by intertwining XML metadata with tags for intervals (CMML) with the video and audio streams into a format called Annodex [2]. However the organization that developed Annodex seems to have stopped supporting it being open-source and removed the public information. The communities about Annodex seems to be gone as well, meaning trying to use it can be complex or near impossible.

3.3 NoSQL Database

One possible way to store the data could be in a NoSQL database. NoSQL is a very broad term though and many kinds of NoSQL databases would not be suitable for this kind of metadata and what we want to do with it since they only allow search on primary keys [4]. We need to be able to search both for a person to find out what videos contain the face and to search for videos to find out what faces they contain. We also need to be able to search for strings and sub-strings of the transcripts and other kinds of metadata.

Furthermore we also need to be able to do more than just the most basic of queries. Among possible other things we at least want to be able to search with AND/OR operations. This leads to the conclusion that if we are to use a NoSQL database it has to be one with a document based data model such as MongoDB. In MongoDB each ”document” is structured similar to the JSON format. MongoDB also has support for some kind of relations and search on secondary keys.

MongoDB and NoSQL databases also demand more RAM memory and disk space in order to achieve the possible increase in performance.

3.3.1 Pros

- Horizontal scaling. One of the primary reasons NoSQL is gaining popularity is because they excel at horizontal scaling, sacrificing consistency in some cases and functionality in order to allow for adding more servers to address higher demand.

- Dynamic database schemas. In a document driven NoSQL DBMS not all documents need to contain the same value-pairs. This leads to high flexibility if the data can change without having to make new schemas.

- Potential higher performance than relational databases.
3.3.2 Cons

- Many NoSQL databases offer less flexibility when it comes to queries. And depending on which data model is chosen some only allow queries on the primary keys.

- Does not conform to ACID-properties (Atomicity, Consistency, Isolation, Durability).

- Demands more memory (physical and primary) than relational databases.

3.4 Relational Database

Perhaps the most intuitive solution would be a relational database. This would result in a system not entirely unfamiliar which could lead to a better result.

Some of the advantages with a strictly relational database approach would be that it would be a uniform solution which could lead to a less complex solution. A relational database is also a proven and tested approach with plenty of tools and support.

It is however unclear how the scalability for a relational database would be and how much it would depend on the design of the tables and relations.

3.4.1 Pros

- Complex queries. Most relational DBMSs allow for very powerful queries with joins and nested queries.

- ACID-properties.

- When done right can make use of natural relations between data to create schemas that are easy to understand.

3.4.2 Cons

- Static data models. After a relational database is set up and the schemas are done it can be a lot of work to change the data models. For some changes the whole database may have to be reset.

- Performance can be an issue with large and complex schemas.
3.5 Conclusion

Some of the main advantages of a NoSQL database solution is not a priority for this project. The flexibility that comes with dynamic schemas does not really matter since the detection programs will generate files with the same structure and containing mainly the same things.

Horizontal scaling is a good property but we are mainly interested in vertical scaling. Being able to search efficiently in large sets of data and not as concerned with many users being able to search at the same time.

Something that we do want is to be able to make many different kinds of queries. A relational database would certainly give us the tools to make as complex queries as we might want. In a NoSQL DBMS it depends on the specific DBMS and it would require more in-depth study of the different alternatives to see if they support the complexity we need. Since the ultimate purpose of this thesis is to provide a way to handle video metadata not only for the given scenario but for a more general platform it is a really good property to be able to do more complex queries than we might need at the moment.

We also want some kinds of relations between faces and the videos. MongoDB allow some basic relations by being able to link to other document, this would probably be enough, but it seems like this approach would remove most of the performance gains we could get by using NoSQL [3].

Another aspect in the decision is that due to my unfamiliarity with NoSQL it is unclear whether I will be able to make a technical demo of that kind of system within the time-frame of this project.

We also looked at which DBMSs conform to the ACID properties (Atomicity, Consistency, Isolation, Durability) which guarantee the reliability of transactions. This is something that most relational DBMSs do but NoSQL seems not to. At least not to the same extent. These properties can be both a strength and a weakness for this project. Since it provides reliability but the performance cost could eventually be too expensive. But for most databases to be useful they need to be reliable.

All of these things combined lead me to believe that the best approach is to choose a relational database.
Chapter 4

TECHNICAL DETAILS

This section will discuss technical details and decisions that were taken for the technical demo that could have a non-trivial impact on the performance of the demo.

4.1 Choice of DBMS

The first restriction on the choice of which RDBMS (Relational Database Management System) to use is that it should be free and preferably open source but could be under some similar license. This usually leads to a choice between PostgreSQL and MySQL. Both are very common and are time-tested and offer a wide range of functionality. MySQL has both a commercial license and a GPL (GNU General Public License) license, PostgreSQL have their own license but it is quite similar to the GPL. Another a little newer option that also fulfill the same requirements is Firebird.

Since it can be assumed that all of these offer enough functionality for our needs the next criterion we will evaluate them on is performance. According to Liang & Lu in their evaluation of relational database management systems from 2010 [5], Firebird and PostgreSQL outperforms MySQL both when inserting and selecting rows of varying sizes. With PostgreSQL seemingly performing the best when inserting and Firebird the best when selecting.

Although insert & select performance is very important for this project the choice will still be PostgreSQL. This is mainly due to lack of knowledge about Firebird. There is also some concern about reliability since it have the best performance but still does not see widespread use [6]. It also seems to suffer from a lack of documentation. It does however have potential to gain more widespread use in the future if some of the uncertainties are resolved.
4.2 Database Schema

The creation of the database schemas is very important for this project since it will determine how easily we can make certain queries. It could also have a significant impact on the performance for selections and insertions. The primary goals is to not have complex dependency relations. We also want to be able to filter as much as possible in the tables before performing joins to avoid having to deal with incredibly large tables, thus we want to keep the most likely queries in mind when designing the schema.

![ER Schema of database tables and their relations](image)

Figure 4.1: ER Schema of database tables and their relations

The schema that will be used going forward can be seen in Figure 4.1, it consists of four tables. These tables are:

**Faces:** The table containing the different faces that have been found in various videos. It consists of an id, a feature vector (a link to an image or a clip that can be used to compare this face against others), a name and additional metadata such as which country the person is from, height etc. For the demo the name will be a single string but for a real application it
might be preferable to have one field for first name and one for surname.

**Videos:** A table containing the various videos we have on the platform, similarly to the faces table it consists of an id, a link to the actual video, a title and additional metadata such as upload date, genre, director etc.

**Face Intervals:** This is a table that links together a face and a video as an interval. As can be seen in Figure 4.1 it has many-to-one relationships to both faces and videos. This means that that one video and face combination may have several face intervals. The table also has a start time and an end time of the interval. This interval represents that a certain face appears in a certain video between these two times.

How good this design is may depend upon the face recognition software to some extent. If the software creates a lot of small intervals this table may become enormous. One reasoning behind this design is that it follows closely to the XML output generated by the face recognition software that can be seen in Appendix A.

This table could also be extended with additional metadata such as mood of the person in the interval or with a description of what the person is doing etc.

This table also contains two additional indices other than the primary key, those extra indices are B-tree indices on the fields vid and fid. They are needed in order to improve the performance of join operations on the table. This is especially important since most queries will have to join this table with another due to the face intervals being in the center of the whole database connecting everything.

**Frame:** The last table contains frames that are linked to a face interval in a many-to-one relationship. It contains information about where in the video the face (or object) is and at what exact timestamp. The purpose of this table is to be able to allow for interactive handling of the face in the video by using it’s position.

The reason it was designed as a separate table is that for many videos/faces we may not want to track the position and then we can simply not create corresponding frame rows. It is also unlikely that we would like to fetch the actual frame metadata for most queries in which cases we can simply not join together the frames with the face interval.

One thing of note is that the Frame table does not contain an index field in order to create a candidate key. Instead the superkey of this relation is a combination of which interval it belongs to and what timestamp (including frame) it contains. This design decision should speed up the process of joining.
this table with a face interval table since it will allow for index scans instead of sequential scans, which are often faster when SELECTING relatively few rows from large sets of data. Now we let the query planner choose which kind of plan to use instead of only having sequential scans.

We could have had a regular index field in combination with an additional index for interval ID but that would lead to a more complex schema and additional disk space and time to keep both indices up-to-date. The only drawback with this approach is that it makes it slightly more difficult to select one individual frame if the user wants to update or delete it for example, since that now has to be done with both an interval ID and a timestamp compared to just an ID.

The timestamp for each frame is on the form:

"HH:MM:SS.MS"

In the timestamp MS represents at what time the frame should appear. For example, with a video running at 10FPS it would be: .0, .1, .2, .3 and so on. For a video running at 25FPS it would be: .00, .04, .08, .12 ... , .92, .96. Another possible way to represent each individual frame would be to simply number them from 1 to the FPS being run.

There aren’t many controversial choices in the schema as it is fairly straightforward. It is also in BCNF (Boyce-Codd Normal Form). I thought about adding an AppearsIn table with a many-to-many relationship between faces and videos as well for quick queries where someone wants to know in what video a person appears, or which people appears in a given video. This approach could help improve performance for some queries, but since we do not know yet how long time it takes to query those things without the extra table it was deemed surplus.

4.3 Queries

The following subsection will discuss the queries and performance issues surrounding them.

4.3.1 Creating Tables

Queries for creating the tables used for the test programs can be found in Appendix B.
4.3.2 Insertion

Since performance of insertion is not a direct concern of this thesis unless it proves fatally slow it will not be discussed much other than that it is done by transactions to ensure that either a full video with all data is inserted or none of it. This ensures that our dependencies are correct and it also improves performance significantly compared to many small commits.

4.3.3 Selection

There are various selection queries of different complexity. Much of the idea of this solution is to fetch information from the database in different steps. For example, when searching for two persons appearing at the same time. The database is first queried for videos with an interval where the two people overlap at some time, and return a list with those videos. If the user then picks one of those videos we fetch the actual times they are overlapping in that video. If one of those intervals is picked, we gather the actual frame positions for both. Thus we can spread out the query-times and make the experience smoother for the user. This approach simulates how an actual use case could be for the given scenario.
Chapter 5

TESTS

The tests performed on the technical demo will be crucial for the results achieved and for the conclusions of this thesis. This chapter will list what tests will be done, how they will be executed and on which hardware.

5.1 Test Machine

All testing was done on a machine with the following specifications:

Operating system: ubuntu 14.04 LTS, 64 bit
Processor: Intel Core i7-4702MQ 2.20GHz x 8
Primary memory: 7.5 GiB
Disk space: 238 GB

Running PostgreSQL version: 9.3.12.

5.2 DBMS-Configuration

There are plenty of ways to optimize database performance with settings in the DBMS, this is not something that have been studied in-depth for this thesis and the testing will begin on a default-configuration of a PostgreSQL DBMS. With the goal that scalability trends will emerge.

5.3 Filling The Database

A program has been written that randomize a video in a model very similar to the XML example in Appendix A. It creates a video of length 100s, it then randomizes between 1 and 15 faces for that video. Each of these faces
appear in a random amount of intervals between 1 and 20. These intervals have a random offset in time from the last interval and a random length, they then fill up every second of the interval with the correct amount of frames (it only fills integer amount of seconds with frames, this is only a prototype). Each second has 10 frames.

Both the mentioned time offsets are between 1 and 10 seconds, with only integer values possible.

The fields whose values will not be used in the tests such as frame coordinates, a video’s actual video link and the feature vector for faces will be filled with dummy values so they will affect the disk space somewhat realistically.

Faces and Videos get a random name that is ”Face: ???” where ??? represents a number. Different videos can have the same title. But if a new face has the same name as one already in the database it will be considered the same face. If this was a real application this would not be the case since two persons can have the same name, but for our prototype this helps us simulate that one person can appear in many different videos. Which should help simulate the various cross-dependencies that may occur. The title/name number is randomized by generating a number between the amount of videos currently in the database and the amount of videos being added in this batch.

Example: If there is 0 videos in the database and you add 10,000 videos, the numbers in the names/titles are between 0 and 9999. If you add an additional 40,000 videos the numbers will be between 10,000 and 49,999. This should ensure a fairly even spread no matter in how large batches the videos are added.

One thing that should impact performance is the ratio between videos, faces, intervals and frames. But since we do not know what ratios these will be like for a real application and since they can vary between videos and especially between different types of videos (action movies versus nature documentaries for example) we will go with this randomized procedure.

5.4 Vertical Scaling Tests

For testing the vertical scaling of the system the database will step by step be filled with more videos. In the first stage the database will be filled with 10,000 videos. Then the database will continue to be filled until it reaches population levels of 50,000, 100,000, 200,000 and finally 300,000 videos. For each population level of videos the disk usage of the database will be mea-
sured and a series of queries will be made and timed. These queries will not be done in the demo but will instead be done directly to the database using the terminal on the local machine. This in an attempt to remove outside disturbances on the query times such as latency.

All tests will be executed via bash-scripts. All PostgreSQL queries that will be used in the tests can be found in Appendix C. In order to further minimize unwanted variance each script will be run ten times and the average will be calculated.

Graphs & figures will then be made for each test comparing the execution time to the database population level in order to identify scalability trends.

5.4.1 Test1 - Finding 10 Face Intervals From Video Title

We will search for 10 face intervals belonging to ten different titles. There can be multiple videos that matches the title we’re searching for. The reason that we only search for 10 intervals instead of all of them is to reduce the varied impact of how many rows are returned.

The script will randomly select 10 video titles by searching for titles containing a number between 0 and the number of videos in the database - 1. There may be multiple videos that matches that search query and if so the first 10 intervals will be returned. The time it takes to get these 100 intervals is measured.

5.4.2 Test2 - Finding 10 intervals where face A and B overlaps

Query the database with the name of two persons and get either no result if they never overlap or between what times they overlap and in which videos the first 10 overlaps happen. This test will randomize 10 pairs of names and search whether these pairs overlap at some time interval in any video. If they do it will return 10 of these intervals with the actual time they overlap. The time it takes to get these 100 intervals is measured.

An important aspect of this test is that it searches for people with names similar to the randomized names. Because in the used scenario a user might for example not know both the first name and the surname of the person they want to find.
5.4.3  Test3 - Find all frames where face A and B overlaps during interval X

Given that the persons A and B overlap between two timestamps in a given video, fetch all frames belonging to that interval. The script will begin with a search similar to the test in 5.4.2 with the difference that only the first interval where each distinct pair matches is chosen. The time it takes to get all frames within these intervals are then measured.

5.4.4 Disk Size

The total disk size of the database is measured by running the commando "\l+" in PostgreSQL. This test is done for all population levels of the database.

For the population level of 100,000 & 200,000 videos, the disk size of each individual relation will also be measured. To see how much of the disk size that comes from indices and those kinds of tables. This was done with the query:

```sql
SELECT nspname || '.' || relname AS "relation",
       pg_size_pretty(pg_relation_size(C.oid)) AS "size"
FROM pg_class C
LEFT JOIN pg_namespace N
  ON (N.oid = C.relnamespace)
WHERE nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY pg_relation_size(C.oid) DESC
LIMIT 20;
```

5.4.5 Insertion Times

Insertion times will be measured for each incremental step. Since it is not of the utmost importance they will not be done in an overly scientific way and should not be cited. As other processes can have been run at the same time. They are mainly measured to get a hunch about what the insertion trends seem to be.
Chapter 6

RESULT

This section will contain all test results and explain them. For a more detailed account of the tests and how they were made see chapter 5.

6.1 Test1

Figure 6.1: Average execution times of Test1 plotted against the number of videos in the database
Test1 measured how long time it took to get 10 face intervals for 10 different people. This was done 10 times and an average was calculated for each population level. By creating a plot for these time - population level pairs it can be deduced that it looks like the trend is something similar to log(n), which can be seen in Figure 6.1. The last point for a population level of 300,000 deviates from the pattern.

There are various reasons why the last point might deviate. One likely reason is that the test machines performance might vary, and since the difference between most of these times are very small it could lead to a noticeable result.
6.2 Test2

Figure 6.2: Average execution times of Test2 plotted against the number of videos in the database

Test2 measured how long time it took to get overlapping intervals between 10 pairs of people. This was done 10 times and an average was calculated for each population level. When the population level is plotted against the time as can be seen in Figure 6.2, it looks to be similar to a log(n) curve since the slope between two points seems to get smaller and smaller for the higher population levels.

This is both the expected result and the desired result.
6.3 Test3

Figure 6.3: Average execution times of Test3 plotted against the number of videos in the database

Test3 measured how long time it took to retrieve all frames that belong to 10 overlapping interval. The test was repeated 10 times and the average value was plotted against the population level of the database, which can be seen in Figure 6.3.

Nothing can really be concluded by looking at the graph. It seems to either hold bad values or the query planner changed how the query was executed between the population levels. Because it is highly unlikely that the query times should increase at first, then decrease for the next population level. The differences in execution time between the minimum point and the maximum point are also very small.
6.4 Disk Space

Figure 6.4: Total disk size of the database in GB plotted against the number of videos in the database

The first test for disk space was that the disk space used by the database was measured for each population level. These points were used to construct a graph. Disk space has a completely linear correlation with population level which can be seen in Figure 6.4.

The second test was to measure what in the database that took up relevant amounts of disk space for two different population levels. The values from these measurements can be seen in Table 6.1.

Two things can be immediately concluded from the table. The first one is that this also seem to have a linear correlation, with every relation almost exactly doubling in size when the population level is doubled. The second thing is that all sizes are almost trivial when compared to the size of the frame relation and the frame index. The second largest table, Face Intervals is merely a few percent of the size of the size of the frame index.
<table>
<thead>
<tr>
<th></th>
<th>100.000</th>
<th>200.000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Videos</td>
<td>100.000</td>
<td>200.000</td>
</tr>
<tr>
<td>Frames</td>
<td>22 GB</td>
<td>43 GB</td>
</tr>
<tr>
<td>Frames PKey</td>
<td>9 GB</td>
<td>18 GB</td>
</tr>
<tr>
<td>Face Intervals</td>
<td>319 MB</td>
<td>638 MB</td>
</tr>
<tr>
<td>Face Intervals PKey</td>
<td>119 MB</td>
<td>238 MB</td>
</tr>
<tr>
<td>Face Intervals Other Keys</td>
<td>270 MB</td>
<td>540 MB</td>
</tr>
</tbody>
</table>

Table 6.1: Table of which relations had the biggest disk size for population levels 100.000 & 200.000 videos

6.5 Insertion Times

The database was filled in incremental steps, the total insertion time for 100.000 videos was calculated as the time it took to fill 50.000 videos, plus the time spent adding another 50.000 videos to reach the total of 100.000. These total times was plotted against the population levels to see insertion trends. The thing of note in Figure 6.5 is that the slope between points get steeper and steeper. It takes more than twice the time to fill from 50K to 100K than from 0 to 50K.
Chapter 7

DISCUSSION

This chapter will discuss the test results and the findings of this thesis in general.

7.1 Test Results

In general the test results were good and proved useful.

7.1.1 Search Times

Out of the three tests on search times two yielded graphs that looked like what was expected and desired. The graph from Test3 looks like no conclusion can be drawn from it at all, other than that the search time was fast for all population levels, but the graph is most likely garbage. There can be many reasons for this. It could be that it is also a log(n) curve but that has already reached the flattened out part at the population level of 10.000 and thus time difference varies mainly with outer factors.

Another thing that impacts search times is how many "hits" the query got. Since each test run makes 10 queries, we reduced this impact by limiting the amount of possibly returned rows per query for test 1 & 2 to 10. This means that for example if test1 makes 10 queries and all get "hits" and return rows and the next test it get 0 "hits". We have redone each test iteration with 0 hits to further reduce this impact. By observing all tests during their execution and looking for anomalies, I do not think this should impact much, since there does not seem to make much difference between 1 hit and 2 or 3 hits, which are the most common ones.
The search time will be affected by the lengths of videos and the number of faces/intervals in each one. The scalability trends should not. If all videos in a data set look like the example videos used in this thesis and one video is 10 times as large, finding the intervals or specific overlaps in that longer video could take a little longer time but searching in one of the others should be roughly the same, and if all videos were much larger all times would be slightly longer but should still have a similar trend.

Another important thing is that it should be possible to optimize the database to get even better results.

7.1.2 Disk Space

The tests about disk space had results that were good. A linear increase in disk space against population level is the best possible result.

The other thing we wanted to find out was how large the indices were to see whether it was a problem to have as many indices as we did. The test showed that all indices and relations are dwarfed in comparison to the ones for frames. Thus it is not a problem to have plenty of indices on the Face Interval relation.

7.1.3 Insertion Times

Insertion times were very slow and had a negative curve meaning that the more videos we add the slower it gets, this is most likely due to the time it takes to update all the indices for each insertion. There are various methods to speed up bulk insertion though. One such method is to remove all indices before bulk insertion and then rebuilding the indices afterwards, this however is not recommended in practical implementation of this system since other users might want to use it during the time the indices are being rebuilt. Another method is to use multi-row insertion or the COPY command from PostgreSQL, which should speed up insertion dramatically, even though it is possible that these methods also have a negative trend.

7.1.4 Indices

Plenty of things have been learned about indices and if one should have a database for per-frame metadata, making decisions about which fields to keep indices on is a key issue. Adding an index on every column that will be scanned by one of the queries will in most cases speed up the query substantially but will cost disk space and lead to slower insertions.
One thing that would most likely have to be used for this kind of solution for a real-world application is more support for text-search. This can be done by adding GIN & GiST indices on the fields videos.title and faces.name. They would allow for returning rows depending on how well the search string matches the indexed field. Compared to my demo where rows are returned if they contain the exact search string somewhere in the complete string.

### 7.1.5 Problems

There are many variables at play when making these kinds of tests and to establish the importance of each one can be very hard. Since our goal was to look at scalability trends we have not laid much focus on the importance of hardware, the importance of changing the input data and other things like that.

A better result could have been achieved by testing even larger data sets but the insertion times limited my ability to do this in this thesis. Disk space would also have been a limiting factor if testing much larger sets.

It would also be interesting to make a comparison looking at what happens with the search times with different ratios between faces, intervals and videos.

### 7.2 Metadata

Since metadata is at the center of this project and the goal of this thesis is for the result to be able to be used for a wide variety of situations where per-frame metadata is desired, here follows a discussion on how the model can be used for other types of metadata.

#### 7.2.1 Face/Object Tracking

One way to expand the model is by adding additional metadata fields to the face intervals table. This could for example be metadata such as mood. This would make it possible to search for every interval where person "Foo Barsson" appears and is happy. But it could also be what kinds of clothes a person has. This kind of metadata expansion by adding fields to the table is suitable if it is likely that most of the videos in the same database will set the metadata for that field. If the additional interval metadata changes a lot
between intervals and videos an alternative solution would be needed. The solution could be to add a field that can be a list of tags for example, which is likely slower to search in but less messy.

One could also expand the frame table with additional fields to include more per-frame metadata, although this should be avoided if possible as it will lead to much larger databases. So unless it is a must to have it at frame level such as the object position it should be kept at interval level or even more preferably as global metadata.

Another thing that should be considered when using per-frame metadata for face/object-tracking, is whether it is entirely necessary to track the position for each frame or if it is possible to get a good result by saving the position for every fifth (or any other number) frame and interpolate to get the points between during playback. It seems likely that for most videos this should result in a fairly good result unless the clip in question have lots of quick cuts while maintaining the same object in view.
7.2.2 Global Metadata

Global metadata for videos can easily be added to this model by expanding the amount of fields for the videos table. The same can be done for faces as well.

7.2.3 Transcript Metadata

A solution that works for transcript metadata should also be able to be kept similar to that of object tracking. By using transcript intervals which contain a start time, end time, which video, which face, what is said and in what language and additional fields for further metadata the existing solution should work well. If one wants to connect these transcripts to frame level it could also be done in the same way; this could be desirable if for example it is important exactly when part of a subtitle should be shown.

7.2.4 Query Flexibility

If one wants to allow more flexibility in the queries that should be very doable for this solution. By allowing for special symbols in the actual platform which would then be pattern matched to create corresponding queries. The only things that should change is that some additional indices need to be made on the new fields. These fields can then be searched. For example if we want to be able to search on which videos where person X appears and is "Homemade" we could put an index on the new metadata field for whether it is "Homemade" or "Professional" and construct a query for it.

7.3 Conclusion

A relational database and more specifically PostgreSQL seems suitable for handling per-frame metadata given that the metadata is fairly static so that the relations and schemas can be designed well. The search-times were short for the tests made and did not seem to be linear (execution time being linearly dependent on the size of the set being searched in) or worse.

The extra indices needed to make it fast took some space but compared to the space taken up by the frames it was fairly trivial and should not be a problem for most applications since disk space is cheap. The large relations were fairly big but this is natural since any system for storing per-frame metadata will end up big. This would likely be an even bigger problem for a NoSQL dbms such as MongoDB though, although the difference probably would not be very big.
One thing of importance when making a real application for this is to have sufficient primary memory for the database since some of the join tables could become huge.

The relational dbms approach is also as mentioned in the pre-study a time-tested solution that conforms to the ACID properties and can be expected to behave as intended and keeping dependencies correct as long as transactions are used properly.

For projects with dynamic per-frame metadata or with plenty of different types of metadata a document based NoSQL approach is likely the better alternative.
Bibliography

[1] What Happens in an Internet Minute?
   Intel
   2013

   CSIRO-ICT, North Ryde, NSW, AUS
   Silvia Pfeiffer, Conrad Parker, Claudia Schremmer
   http://dl.acm.org.proxy.ub.umu.se/citation.cfm?doid=973264.973279
   2003

   Alfred Wester, Olof Fredriksson
   Blekinge Tekniska Högskola
   2012

[4] Scalable SQL and NoSQL Data Stores
   Rick Cattell
   ACM SIGMOD Record: Volume 39 Issue 4, December 2010
   id=1978919&acc=ACTIVE%20SERVICE&key=74F7687761D7AE37%2E7E1D6D73D659A8A5%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=591488244&CFTOKEN=90401905&_acm__=1458209573_4ac3e8db51bd8e2ff711cf5d0486aea5
Xing Liang, Yongyu Lu
Halmstad University
2010

[6] DB-Engines Ranking of Relational DBMS
DB-Engines
http://db-engines.com/en/ranking/relational+dbms
2016, March
Appendix
Appendix A

XML example from face detection program

```
<ClusteringReport>
  <face id="1">
    <featureVector>...</featureVector>
    <representativeFrame>30</representativeFrame>
    <timespan start="29@25" end="639@25">
      <frame time = "29">
        <point x="0.65" y="0.21"/>
        <point x="0.88" y="0.43"/>
      </frame>
      ...
    </timespan>
    <timespan start="800@25" end="900@25">
      <frame time =800>
        <point x="..." y="...">
          <point .. />
        </point>
        ...
        ...
      </frame>
    </timespan>
  </face>
  <face id="2">
```
...</ClusteringReport>
Appendix B

PostgreSQL commands for creating the database tables

CREATE TABLE videos (    
  video_id SERIAL PRIMARY KEY,    
  title VARCHAR(200),    
  video_link VARCHAR(400)  
);

CREATE TABLE faces (    
  face_id SERIAL PRIMARY KEY,    
  name VARCHAR(200),    
  features VARCHAR(400)  
);

CREATE TABLE face_intervals (    
  interval_id SERIAL PRIMARY KEY,    
  fid INTEGER REFERENCES faces (face_id),    
  vid INTEGER REFERENCES videos (video_id),    
  starttime TIME,    
  endtime TIME  
);

CREATE INDEX face_intervals_fid_idx ON    
  face_intervals USING btree (fid);    
CREATE INDEX face_intervals_vid_idx ON    
  face_intervals USING btree (vid);
CREATE TABLE frame (  
    iid INTEGER REFERENCES face_intervals(  
        interval_id) NOT NULL,  
    coords BOX,  
    timestamp TIME NOT NULL,  
    PRIMARY KEY(iid, timestamp)  
);
Appendix C

PostgreSQL queries for testing database performance
C.1 Test1 - Bash Script

#!/bin/bash

set -e
set -u

# Set these environmental variables to override them,
# but they have safe defaults.
export PGHOST=${PGHOST-localhost}
export PGPORT=${PGPORT-5432}
export PGDATABASE=${PGDATABASE-thosun}
export PGUSER=${PGUSER-test_user}
export PGPASSWORD=${PGPASSWORD-123}

RUN_PSQL="psql -X --set AUTOCOMMIT=off --set
ON_ERROR_STOP=on "

declare -a numbers=()
for i in {1..10}
do
    numbers[$i]=[$RANDOM % $1]
done

START=$(date +%s.%N)

for i in {1..10}
do
    ${RUN_PSQL} <<SQL
SELECT vid, title, fid, faces.name, starttime, endtime
    FROM (videos JOIN face_intervals ON video_id = vid) JOIN faces ON face_id = fid
    WHERE title LIKE '%${numbers[$i]}%' LIMIT 10;
rollback;
SQL
done
dur=$(( echo "$(date +%s.%N) - $START" | bc ); \ 
LC_NUMERIC="en_US.UTF-8" printf "Execution time: %.6f seconds\n" $dur
C.2 Test2 - Bash Script

#!/bin/bash

set -e
set -u

# Set these environmental variables to override them
# but they have safe defaults.
export PGHOST=${PGHOST-localhost}
export PGPORT=${PGPORT-5432}
export PGDATABASE=${PGDATABASE-thosun}
export PGUSER=${PGUSER-test_user}
export PGPASSWORD=${PGPASSWORD-123}

RUN_PSQL='psql -X --set AUTOCOMMIT=off --set ON_ERROR_STOP=on'

declare -a face1=()
declare -a face2=()
for i in {1..10}
do
  face1[$i]=$[ RANDOM % $1]
  face2[$i]=$[ RANDOM % $1]
done

START=$(date +%s.%N)

for i in {1..10}
do
  ${RUN_PSQL} <<SQL
SELECT fid, fid2, f1.name, name2, vid, interval_id, iid2, GREATEST(starttime, st2), LEAST(endtime, et2)
FROM (faces JOIN face_intervals ON (faces. face_id = face_intervals.fid)) AS f1,
  (SELECT fid as fid2, name as name2, vid as vid2, interval_id as iid2, starttime as st2, endtime as et2
SQL
FROM (faces JOIN face_intervals ON (faces.face_id = face_intervals.fid))
WHERE faces.name LIKE '%${face2[$i]}%') AS f2
WHERE f1.name LIKE '%${face1[$i]}%'
  AND f1.vid = f2.vid2
  AND f1.starttime < f2.et2
  AND f2.st2 < f1.endtime) LIMIT 10;
rollback;
SQL
done

dur=$(echo "$(date +%s.%N) - $START" | bc); \
LC_NUMERIC="en_US.UTF-8" printf "Execution time: %.6f seconds\n" $dur
C.3 Test3 - Bash Script

#!/bin/bash

set -e
set -u

# Set these environmental variables to override them, but they have safe defaults.
export PGHOST=${PGHOST-localhost}
export PGPORT=${PGPORT-5432}
export PGDATABASE=${PGDATABASE-thosun}
export PGUSER=${PGUSER-test_user}
export PGPASSWORD=${PGPASSWORD-123}

RUN_PSQL=psql -X --set AUTOCOMMIT=off --set ON_ERROR_STOP=on --no-align -t

declare -a face1=()
declare -a face2=()
declare -a iid1=()
declare -a iid2=()
declare -a st=()
declare -a et=()

OLDIFS=$IFS

for i in {1..10}
do
  face1[$i]=$[ RANDOM % $1]
  face2[$i]=$[ RANDOM % $1]
done

for i in {1..10}
do
  result=$(RUN_PSQL) <<SQL
SELECT interval_id, iid2, GREATEST(starttime, st2), LEAST(endtime, et2)
FROM (faces JOIN face_intervals ON (faces.
SELECT fid as fid2, name as name2, vid as vid2, interval_id as iid2, starttime as st2, endtime as et2
FROM (faces JOIN
    face_intervals ON (faces.
    face_id = face_intervals.
    fid))
WHERE faces.name LIKE '%${ face2[$i]}%') AS f2
WHERE f1.name LIKE '%${face1[$i]}%'
    AND f1. vid = f2. vid2
    AND (f1. starttime < f2. et2
    AND f2. st2 < f1.endtime) LIMIT 1;

SQL
)
IFS='|' read -r iid1[$i] iid2[$i] st[$i] et[$i] <<< "$result"
done
IFS=$OLDIFS

for i in {1..10}
do
if [ -z "${iid1[$i]}" ]; then
    iid1[$i]="0"
fi
if [ -z "${iid2[$i]}" ]; then
    iid2[$i]="0"
fi
if [ -z "${st[$i]}" ]; then
    st[$i]="00:00:00"
fi
if [ -z "${et[$i]}" ]; then
    et[$i]="00:00:00"
fi
done

RUN_PSQL="psql -X --set AUTOCOMMIT=off --set
    ON_ERROR_STOP=on"
START=$(date +%s.%N)
for i in {1..10}
do
${RUN_PSQL} <<SQL
SELECT frame.timestamp, coords, fid FROM
  face_intervals JOIN frame on interval_id = iid
  WHERE (interval_id = '${iid1[$i]}') OR
      interval_id = '${iid2[$i]}') AND frame.
timestamp BETWEEN '${st[$i]} and '${et[$i]}';
ROLLBACK;
SQL
done

dur=$(echo "$(date +%s.%N) - $START" | bc); \nLC_NUMERIC="en_US.UTF-8" printf "Execution time: %.6f seconds\n" $dur