Database Paradigms for Recordings Management

Eric Ottosson, Suraj Alok Sinha

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-21
Database Paradigms for Recordings Management

Eric Ottosson  
eric.ottosson@gmail.com  

Suraj Alok Sinha  
sinha.suraj@outlook.com

June 15, 2015

Master’s thesis work carried out at Axis Communications AB.

Supervisors: Ann-Sofie Rase, ann-sofie.rase@axis.com  
Karin Hedlund, karin.hedlund@axis.com  
Flavius Gruian, flavius.gruiian@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se
Abstract

The relational database has long been considered the de facto standard for managing data in software applications. Today, a need for more scalable, flexible and distributed software solutions has led to the development of NoSQL database technologies that aim to replace the relational database in applications where such features are needed.

In this thesis we have investigated the potential benefits of replacing SQLite, the database used by Axis Communications to manage recordings in their camera products, with a “Not only SQL” (NoSQL) database in an embedded camera system. To evaluate performance, test cases to measure execution times and resource consumption for database operations, based on important functionality in Axis’ storage solution, were designed.

In the end the Embedded JSON Database Engine (EJDB) document database was identified. EJDB was found to be more efficient than SQLite at creating, updating and removing records. It was, however, less efficient when performing queries based on conditional operators.

Keywords: Databases, Embedded Systems, NoSQL, Relational Databases, Document Databases, SQLite, EJDB
We would like to give a special thanks to our supervisors at Axis Communications, Ann-Sofie Rase and Karin Hedlund for their guidance throughout the work, and for giving us the opportunity to conduct this thesis. We also want to thank our supervisor Flavius Gruian at Lund University for providing feedback and direction throughout the work. Finally, a big thank you to the Media department at Axis Communications for all your support and help during the thesis.
## Contents

1 Introduction 11
   1.1 Problem statement ........................................... 11
   1.2 Evaluation process ........................................... 12
   1.3 Report outline ............................................... 12
   1.4 Contributions ............................................... 13

2 Background 15
   2.1 Databases in embedded systems ................................ 15
       2.1.1 Database configurations .................................. 16
   2.2 Managing recordings .......................................... 16
       2.2.1 Recordings information ................................... 17
       2.2.2 Metadata ................................................ 17
   2.3 Relational databases .......................................... 18
       2.3.1 Data structure ........................................... 18
       2.3.2 Column-oriented databases ................................ 20
       2.3.3 NewSQL ................................................ 21
   2.4 NoSQL databases ............................................. 21
       2.4.1 Key-value databases ...................................... 21
       2.4.2 Document databases ...................................... 22
       2.4.3 Column-family databases .................................. 23
       2.4.4 Object databases ......................................... 24
       2.4.5 Graph databases ......................................... 25

3 Axis’ storage solution 29
   3.1 Entity-relationship model .................................... 29
       3.1.1 Entities .................................................. 29
       3.1.2 Model properties ......................................... 31
   3.2 Accessing recordings information ................................ 31
       3.2.1 Inserting and updating recordings ..................... 32
       3.2.2 Retrieving recordings, events and sources .......... 32
       3.2.3 Removing recordings .................................... 33
       3.2.4 Recording playback ..................................... 34
## Discussion

8.1 EJDB and SQLite execution times ........................................ 73
  8.1.1 Conditional operations ........................................... 73
  8.1.2 Impact of the SD card ............................................. 74
  8.1.3 Remove recordings ............................................... 74

8.2 EJDB and SQLite resource consumption ............................... 74
  8.2.1 EJDB transactions and durability ................................ 75

8.3 Increasing performance in EJDB with indices ...................... 75

8.4 Building on Axis’ storage solution .................................. 76

8.5 Selecting a candidate .................................................. 76

## Conclusion

9.1 Future Work ............................................................. 80

### Bibliography

81

### Appendix A  NoSQL timeline

89

### Appendix B  Cross-compiling EJDB

91

### Appendix C  Resource consumption plots

93
Acronyms

**ACID** Atomicity, Consistency, Isolation and Durability. 17, 42

**CAD** Computer Aided Design. 20

**DBMS** Database Management System. 12

**ER model** entity-relationship model. 14, 22, 25, 27, 30, 32, 35–38, 41, 43, 44, 46, 72

**GUI** Graphical User Interface. 71

**JNI** Java Native Interface. 34

**JVM** Java Virtual Machine. 34

**MKV** Matroska Multimedia Container. 26

**NoSQL** “Not only SQL”. 7, 9, 75

**OLTP** Online Transaction Processing. 17

**OODB** Object Oriented Database. 20, 21

**RDBMS** Relational Database Management System. 17, 32, 72

**SQL** Structured Query Language. 7, 14, 28, 29, 37, 38, 46, 51, 70
Chapter 1

Introduction

The relational database paradigm was introduced by Codd in 1970 and models character-based data in terms of attributes and records, translated to columns and rows in a table \[1\]. Since its creation, this paradigm has been the de facto standard amongst a wide variety of applications. Today the paradigm can be found in both large web services such as Wikipedia \[2\], as well as in much smaller systems such as credit cards \[3\].

Relational databases have been used extensively during past decades. However, increasing amounts of dynamic and unstructured data (big data \[4\]) has been driving the need for more easily scalable database technologies \[5\].

“Not only SQL” (NoSQL) is the name for a relatively new group of databases that are built on top of paradigms, such as graphs \[6\], documents \[7, 8\], column families \[9\] and key-value pairs \[10\]. These databases do not rely on the Structured Query Language (SQL) found in relational databases, and provide new solutions to manage the increasingly larger and more complex data found in today’s applications. An overview of the timeline for the appearance of the mentioned paradigms can be found in Appendix A.

The number of surveillance cameras used around the world is constantly increasing \[11\]. This is also true for the amount of information that is captured by such cameras. As the computer industry has evolved, cheaper hardware and larger storage devices have made it possible to produce more powerful cameras. Today, cameras are capable of storing several weeks, or even months, of continuous recordings.

Axis Communications \[12\] is an IT company that offer network video solutions for both private and professional installations. Since their earliest camera products, Axis has used relational databases to manage the recordings for their cameras.

As a company, Axis strives to provide modern security solutions and have high requirements on their products. For Axis, this means that they constantly look for ways to improve upon their products. Thus, the new paradigms introduced with NoSQL motivates them to research these, to see whether they are usable in their camera products.

1.1 Problem statement

We define the following purpose and goal for this thesis:
The purpose of the thesis is to examine the use of a relational database for managing recordings information in a surveillance camera, and to evaluate the potential benefits (faster execution times, lower resource consumption etc.) when moving from a relational paradigm to new paradigms for managing recordings information.

The goal of the thesis is to provide a technical analysis of a relational database used in an embedded camera system, and to compare it against an alternative database paradigm.

1.2 Evaluation process

To accomplish the purpose and goal, we intend to perform a study of the database solution used in one of Axis’ cameras for managing recordings information. This solution will be compared against a new database, employing an alternative paradigm, selected through a process which makes it suitable for use in Axis’ cameras.

Initially we need to perform a pre-study to attain knowledge about available database technologies and how they differ. Axis’ storage solution will then be analysed to better understand what information is needed for recordings management. A survey of available databases is performed in parallel, and a set of requirements is defined. These requirements are then used to aid the identification of potential database candidates for the embedded camera system. Finally, one candidate is selected and evaluated against the database currently used by Axis in their storage solution, using a set of test cases. This whole process is depicted in Figure 1.1.

![Figure 1.1: The process used for finding, and evaluating a database.](image)

1.3 Report outline

The rest of this thesis is structured as follows. In chapter 2 we introduce the theoretical background necessary to understand our analysis. In chapter 3 we describe Axis’ current storage solution and how it is used. We also provide a quick evaluation of the data model
1.4 Contributions

The work has been divided equally between the two authors Eric and Suraj. In the earlier parts of the process Suraj worked more with researching database paradigms while Eric analyzed how Axis use their database. A joint effort was made to design a benchmark for the testing platform, and Eric implemented the test cases for the relational database while Suraj implemented the test cases for the chosen database.

As the thesis progressed both authors have spent time working on every part of the report and have worked together to analyze and evaluate Axis’ storage solution, as well as the different paradigms. Therefore, all discussions and conclusions are the result of both authors work.
Chapter 2

Background

In this chapter we will give a brief introduction to the theoretical parts of the thesis. It contains a short introduction to databases in embedded systems, where different database configurations are briefly described. An overview of recordings data is given and its important characteristics are described. The last part of this chapter introduces the most prevalent database paradigms that can be found today.

2.1 Databases in embedded systems

An embedded system is a computer system, which is designed with a specific function in mind, and it is generally contained within a larger mechanical or electrical system. These types of systems are increasingly more common and appear in a wide variety of applications, such as washing machines, MP3 players, automobiles, among others.

It is becoming common for embedded systems to store and manage data, either collected from their operational environment or provided during configuration. In many cases it is due to increasing complexity in application design, but sometimes the application does not have to be very complex in order to see the potential benefits of data management. For example, smart cards employ relational databases to store credit card information [3].

Compared to full sized computer systems, databases face some challenges when running on embedded systems, because embedded systems usually have limited memory, storage and available processing power. Therefore, memory footprint, file size and processor utilization play an important role and can have a large impact on the performance of the system. This is becoming less of a problem as hardware becomes cheaper and are equipped with more powerful hardware configurations.

Another challenge is that embedded systems in many cases are deployed out of reach for easy maintenance. In such situations it is important that databases for embedded systems are self-managed and capable of backing up data, recovering from failures etc. without user interaction.
2. Background

2.1.1 Database configurations

Databases come in many shapes and forms, but every available Database Management System (DBMS) can be classified into two configurations: either as a server/client system, or as a software library integrated into an application.

**Server/client** This configuration appears when the DBMS is separated from the application being developed [13]. A server will run the DBMS as a separate process, and the client will communicate to send requests and receive responses. There are two hardware configurations that can be used for a server/client system: either the server and client run on the same system in parallel and share resources, or they have separate hardware with their own resources. The server/client configuration is often encountered in multi-user applications, such as web services, but less so in embedded systems. This is mainly because embedded systems often run in isolation without network communication, hence the server and client has to run on the same hardware. There is a potential for widespread use of the server/client configuration as hardware becomes cheaper and the need for centralized or distributed (and cloud) storage in embedded systems increases [14].

**Library** This configuration appears when the DBMS is integrated into the application being developed. By including the DBMS in the application, using a library or source code, it is possible to remove all the overhead produced by a server/client configuration running on the same hardware. Hence, the library configuration is more suitable for embedded systems where the overhead introduced by a server could affect performance.

2.2 Managing recordings

When one thinks of recordings in a camera, one generally thinks of video and audio information, called **footage**, which can be viewed and/or heard. While being a vital component, this type of information is disregarded in this thesis. Instead, only non-footage information, **recordings information** is regarded.

In our study of different database paradigms, we define recordings information to mean the relevant information when capturing and managing footage. This information can be divided into two categories:

1. Details about the footage (i.e. location, which event triggered the recording, which camera captured the footage etc.)

2. Analytics data, also known as **metadata**

A quick note about footage is that it can be stored in two different ways: either kept as a single file, or as a collection of smaller files (which are referred to as **blocks**). The benefit of using a single file is that only one file has to be managed instead of multiple files, but storing footage in multiple files has its own benefits. The footage attains a higher degree of durability, as corruption of data can be limited to parts of the recording instead of the whole. It also becomes possible to retrieve sections of the footage, lowering the amount of bandwidth, without any need to process it in advance.
2.2 Managing recordings

2.2.1 Recordings information

The footage on its own does not facilitate any kind of management. For a surveillance system with many cameras, and large quantities of footage, it is important that footage can be categorized and searched through quickly in order to find the information one is looking for. Based on Axis’ storage solution we define the following properties for such information.

**When was the footage captured** This can be used for filtering footage based on temporal information. Temporal information is represented by storing the date, start time and stop time of the footage.

**Where was the footage captured** In a multi-camera setup, it can be beneficial to know which camera captured the recording, as well as its location (physical placement of the camera). In large installations this information will make it easier for users to search footage.

**What triggered the camera to capture footage** This information is often referred to as events. An event is used to define a trigger that will make the camera perform some action. Depending on the complexity and scope of the surveillance system the action could be anything from starting a recording, to contacting emergency services.

**The quality of the footage** Quality information is important as situations may arise where lower quality footage would lack the details necessary to perform a task. For example, poor quality may make it impossible to identify a car’s license plate number. Quality is most often measured using resolution, but other measurements are possible too, such as bitrate.

**Recording type** A type is a tag which can be used to categorize recordings. Categories could for example be “continuous”-, “scheduled”- or “triggered” recordings, among others. A continuous recording captures footage continuously, a scheduled recording records footage based on a start- and stop time, and a triggered recording captures footage as a result of an event. The information is interesting from a more analytical perspective, where it can provide useful statistics. For example, how much footage was recorded through events.

**The size and storage location of the footage** The importance of the size and storage location of the footage depends highly on what type of system is used to capture it. In an embedded camera system with local storage, the type of system we are concerned with, size and location information is helpful for managing the limited amount of storage space. For example, it is possible to write an algorithm which can identify and discard unimportant footage to make room for new.

2.2.2 Metadata

Metadata specifies what is happening in the recording. It can be useful if, for instance, one wants to identify whether someone forgot their luggage, or find when a car parked in a parking lot. These kinds of searches are impossible using only the previously specified information, unless we have a massive database of events and triggers.

If content awareness is featured in the system, based on the scenarios above the following information is of interest:
The scene and environment  The information about scene and environment is important to establish a context. In the previous example with the car it would tell us that the footage is of a parking lot, and that there are parking spaces where cars can park.

Objects in the scene  The point of having a system that manages metadata is to identify and analyze objects. An object is anything which can move and is not part of the scene. In the example with the parking lot, a car would be an object with some attributes such as license plate number, trajectory (movement pattern), make and model.

Events taking place in the scene  Events describe what is happening in the scene, for example a robbery or an explosion. The relationship between events and objects is that events describes the actions of objects. An event could, for example, be a car parking in a parking space, or a car starting and leaving a parking lot. In these cases the event only describes the parking of the car and when the car leaves the scene, it does not incorporate the movement of the objects.

Event information is a powerful tool for searching and investigating footage.

2.3  Relational databases

All relational databases use the SQL to perform operations on the information stored in the database. The data stored in a relational database is often called structured data because all the entities belonging to the same group (entities that are stored in the same table) have a fixed order, and number, of attributes.

2.3.1  Data structure

In relational databases, data is organized into one or more tables [15, p.18]. A table consists of a number of rows and columns. Each row represents one entity in the table, called a tuple, and each column represents an attribute of the entity. The attributes must have names, such as “name” or “phone number”, describing what kind of values they contain. The names of the tables together with row attributes define the schema, or the Entity-relationship model (ER model), of the data structure [15, p.19]. Figure 2.1 depicts what a table looks like in a relational database. An example of a relational ER model, depicting a blog post, is depicted in Figure 2.2.

Primary and foreign keys

In the relational model there are several constraints that can be placed on the stored data. The two most important of these constraints are primary and foreign key-constraints [15, p.311].

Primary keys are represented using unique values, stored as special attributes, in each tuple. They are used to reference tuples across tables, and as such have a large effect on the overall design of the schema. For example, a newborn in Sweden is assigned a social security number which would act as a good primary key for that person.

A foreign key is represented by a field (or a collection of fields) in one table that uniquely identifies a tuple in another table. These keys are used to maintain consistency across the database by making it impossible to add tuples that violate key-constraints. By
2.3 Relational databases

**Figure 2.1:** Relational database terminology.

**Figure 2.2:** Modeling blog posts in a relational database.

using the concept of foreign keys it is also guaranteed that we have referential integrity, i.e. there are no invalid links in the database [15 p.59].

**Normalization**

One strongly emphasized concept when building relational models is the concept of “normalization”. Normalization tackles two problems which are often encountered in badly designed relational models, namely *redundancy* and *anomalies* [1 p.86].

**Redundancy** Information is repeated, unnecessarily, in several tuples.

**Update Anomalies** If the same value is used in multiple tuples, but only updated in some. For example, let two tuples represent cars manufactured by Volvo. If it turns out that it was in fact Saab who manufactured the vehicles, it would now require two separate operations to update each tuple.

**Deletion Anomalies** If information about different entities are found in the same tuple, and one entity is removed, information about the other may be lost. For example, in a table “Movies” there might be a tuple containing the name, length and star
of the movie. If the star is to be deleted, and this is the only entry for the movie, information about length would be lost.

Redundancy and update anomalies are countered by extracting the redundant, or problematic, parts of tuples into separate tables. For example, if foreign keys were used to reference a table which contained car manufacturers, both the redundancy and anomaly in the above example disappear. As no information is repeated there is no redundancy, and as foreign keys are used a single update in the manufacturer table updates the manufacturer for both tuples.

Deletion anomalies are countered by simply separating problematic tables into several smaller tables.

Relationships in the relational database

Relationships are associations between tables, and there are three different types of relationships that are supported by relational databases. To exemplify these relationships a simple blog service is imagined. In this service a user (author) can have one blog, in which he can post multiple posts. Each post may have no, or several, comments created by other authors. Finally each post may be tagged with an indefinite amount of tags by its author.

**One-to-one** A one to one relation only has one record on either side of the relationship. For example, there is a one-to-one relation between a blog and its author.

**One-to-many** In a one to many relation, a record is related to many different records in the other table. An example of this kind of relation is the relation between a blog and its comments.

**Many-to-many** In a many to many relation, each record in both tables can be related to any number of records in the other table. An additional table is required to store many-to-many relationships, hence this kind of relation has some overhead [16]. An example of this kind of relation is the relation between a tag and a blog post. A tag can be associated with many different posts and a post can have many tags.

2.3.2 Column-oriented databases

The concept of column-oriented databases appeared in the 1970s, when the relational model was developed, but it was not until 2000s that column-oriented databases became popular [17].

Column-oriented databases differ from traditional relational databases in how they store information on disk. In traditional relational databases (row-oriented) data is stored in row major order [17], i.e. all the attributes of a row are stored together as a data object. On the contrary, in a column-oriented database all the data in a column is stored together on disk. Thus, in a column-oriented database, the tables can be perceived as being vertically partitioned [17].

Column-oriented databases are a better alternative for applications that execute queries that only need to access a subset of the columns of a table. Hence, it is the access pattern that determines whether a column-oriented approach is suitable or not. If a record tuple is to be fetched from a hard drive, a column-oriented database will have to seek several times to access the record because columns are not stored together [17].
There are few commercial implementations of the column-oriented database, many of them are based on one of the following prototypes: MonetDB \cite{18}, VectorWise \cite{19} and C-Store \cite{20}.

### 2.3.3 NewSQL

The term NewSQL was first mentioned in 2011 by Matt Aslett from 451 Research \cite{21}. NewSQL is a modern Relational Database Management System (RDBMS) that strives to provide scalability for the relational paradigm. This is useful when a RDBMS has to manage a large collection of data that is constantly evolving, called big data \cite{4}. NewSQL and big data is most often encountered in applications which rely on Online Transaction Processing (OLTP). These applications manage transactions over the internet, and are characterized by having many users performing short transactions.

OLTP is an area extensively dependent on RDBMS because of its need for transactions with Atomicity, Consistency, Isolation and Durability (ACID) properties. New OLTP applications have higher performance requirements and NewSQL databases emerged to satisfy their needs \cite{22}. There are some NoSQL alternatives with ACID transactions, but it is not an easy task to migrate from RDBMS to a NoSQL database as the data models are different.

NewSQL databases are able to achieve better performance than traditional RDBMS by applying techniques used in NoSQL databases, such as column-oriented data storage and distributed architectures, among others \cite{22}.

### 2.4 NoSQL databases

NoSQL stands for “Not only SQL” and was coined by Carlo Strozzi in 1998 for his own database, which was a relational database without a SQL interface. The NoSQL movement began in 2009 with a conference that presented a collection of new databases \cite{23}.

NoSQL databases were designed to provide a scalable storage solution for semi-structured data \cite{24}. Semi-structured data is not suitable for relational database because all the entities in the same group may have different number of attributes and the order of these attributes may vary as well. Semi-structured data can also be subjected to rapid change. For example, it is possible that new attributes are added during the execution of the application.

NoSQL databases are built on paradigms other than the relational, such as key-values, documents, column-families, objects and graphs. In this list of different models the first three models are built on the concept of aggregate stores. An aggregate is a structure that contains data that is closely related and accessed together as a unit. In a key-value database, the value is the aggregate. In a document database, the document is the aggregate. For column-family databases each column-family represents an aggregate \cite{25}.

#### 2.4.1 Key-value databases

The key-value model has its roots in Amazon’s Dynamo database \cite{26}. The concept of key-value has been around for a long time, but it was Amazon that started using it for persistent storage around 2007 in its Dynamo database. The key-value model is the simplest model in the NoSQL family.
Data structure

Key-value databases are built on the concept of associative arrays, which is an abstract data type that contains a collection of (key, value) pairs. In databases based on the key-value model, data is stored as values and the associated key is used to access the data. Key-value databases are designed for applications that only need to perform key based lookup, since it is not possible to access the value by other means.

Only keys can be used to access the data stored in the database, thus support for queries is limited compared to other data models. The data model does not support relationships among the entities stored in the database. Hence, additional infrastructure would be needed to handle the relationships [5, p.191].

Figure 2.3 depicts one possible way to represent relationships between entities in a key-value database. In this model, all the information related to a blog post is stored as a unit. Unlike in the relational model, shown in Figure 2.2, embedding information is necessary for key-value databases because of their simple data model.

![Figure 2.3: Modeling blog posts in a key-value database.](image)

2.4.2 Document databases

Document databases emerged as an option to relational databases, mainly aimed towards web-oriented and distributed applications. The purpose of the database was to provide a scalable storage solution for semi-structured data that web services started to demand [7].

Data structure

In a document database data is stored as documents. Each document can contain semi-structured data, and the documents are retrieved as single units (one cannot access a single field without retrieving the whole document). There is a unique identifier associated with each document, which facilitates access to the document. The structure of a document database makes it possible to draw a parallel between it and the key-value database. A (key, value) pair in the key-value model corresponds to an (ID, document) pair and therefore document databases can be considered a subclass of the key-value database, but they process the stored data differently. In a key-value database the data is opaque to the database. However, in a document database the internal structure of the data is known to the database. Thus, it is possible for a document database to support value based queries.
(retrieve documents based on their content). One could also consider each document in the database as a separate key-value store since each document is composed of fields that represent (key, value) pairs.

As mentioned, each document has a unique id that can be used to retrieve the document, provided that the application using the database knows the IDs of the documents it needs. But it is also possible to access documents using value based queries. Since document databases have support for value based queries, they also support indexing on values stored in a document. Indexing is done to support fast retrieval of relevant documents \[5, \text{p.187}\]. Indexed fields are stored in a variants of B trees, known as B+ trees \[15, \text{p. 633}\], which provide \(O(\log n)\) lookup times for records.

Figure 2.4 depicts how a blog post can be modeled in a document database. Notice that comments and tags are stored inside the “post” document. This illustrates the use of embedded data structures, a solution which circumvents the lack of join operations. Entities that have no relevance on their own are good candidates for being embedded.

![Figure 2.4: Modeling blog posts in a document database.](image)

### 2.4.3 Column-family databases

Column-family databases are based on Google’s BigTable \[5\], whose development began in 2004. The data model of a column-family database is schema less, which makes it possible to store semi-structured data, but at the same time it is a tabular database (like the relational model). Apache Cassandra is the most popular column-family database \[27\], it was initially developed by Facebook.

**Data structure**

The basic building block in the column-family data model is a column, which is simply a (key, value) pair. The columns with related data are grouped together to form a column family \[28\]. The conceptual view of this database is that of a table. Each row in the table contains several columns and the number of columns can vary from one row to another. A row in a column family database represents an entity (e.g. it contains everything about an author) and each row has a unique row key associated with it.
As previously stated, a column family database is an aggregate database like document- and key-value databases. However, it has a more expressive data model because it is possible to access parts of an entity stored in the database [5, p.194]. For example, it is possible to only retrieve personal information about “User 1” in Figure 2.5. Since a column family database is still an aggregate store with no support for join operation, there is a need for external application infrastructure such as MapReduce [29] to manage relationships among entities.

Figure 2.5 depicts how blog posts can be stored in a column-family database. There are two tables, each with two rows. “User 1”, “User 2”, “Blog post 1” and “Blog post 2” are the row keys. “Content” and “Personal information” are column families because they contain several columns. “Post” and “Tags” are columns since they only contain one column each.

![Figure 2.5: Modeling blog posts in a column-family database.](image)

## 2.4.4 Object databases

Object Oriented Database (OODB) started to arrive in the mid-80s [30]. A need for a database that was capable of handling complex data from applications such as Computer Aided Design (CAD) software, among others, was the driving force [30]. In the OODB complex data is represented by objects, defined in a object-oriented programming language.

An object has both attributes and methods associated with it. Hence, objects can contain both data and executable code. This makes them complex compared to primitive data types, such as strings and integers. Another purpose of this model is to preserve the meaning of the data when it is stored in the database. The object in the application and in the database have the same structure and contain the same information. However, the object-oriented model has a drawback: there is no common data model for OODB, like the one we have for relational databases [51]. Hence, many definitions exists for OODB. Bagui defines it as: “OODB is a database that integrates object orientation with database capabilities” [30].

### Data structure

In the object-oriented data model, objects are the fundamental unit of storage. Each object in an OODB has a unique identifier, called System-Defined Identifier. Apart from the identifier there are three other characteristics that describe an object:


- A unique name (optional).
- The object lifetime
- The object structure (attributes have values that can be specified using a constructor).

In OODB there are classes with similar functionality as classes in Java – or any other object-oriented language. Objects are instances of these classes, and like in object-oriented languages we also have support for inheritance and class hierarchies [32]. Multiple inheritance is not a common feature for OODB, but there are systems supporting this concept, e.g. ORION [33, p.165].

Like in object-oriented languages, OODB also support objects with properties and methods, as well as embedding objects into other objects. It also has support for all the different relationships that are possible between objects in an object-oriented programming language [32]. The support simplifies the process of managing relationships in an OODB, compared to the relational databases.

At the same time the object model is inflexible as it is not possible to modify the database dynamically. For example, it is not possible to add a new attribute to an already existing object [30]. If new attributes are added to the class definition, the class must be recompiled for the change to take effect.

Figure 2.6 depicts how blog posts can be managed in an object-oriented model. Instances of the classes shown in the figure will be kept in the database.

![Figure 2.6: Modeling blog posts in an object-oriented database, using UML notation.](image)

2.4.5 Graph databases

The term *graph* was first coined by Euler in 1735 in a paper named Seven Bridges of Königsberg [34]. There is a theory dedicated to graphs, called Graph Theory.

Graphs can be found in many places, for example when modeling a relational database. The first step when creating a relational model is usually to create an ER model which is later transformed into tables. Graphs are also used to model transportation networks, communication networks, social networks, among others.
Neo4j \cite{6} is one of the most popular graph databases \cite{27} and it was initially released in 2007. OrientDB \cite{35} is another popular database that supports the graph model and uses SQL as its query language.

**Data Structure**

The name graph databases reveals how data is represented, i.e. through nodes and edges. Nodes are used to represent entities and edges are used to describe the relationship between the entities. Thus, a graph represents pairwise relationships between entities, and the edges describing the relation can be both directed and undirected. If a graph is directed then each edge has a direction, and both nodes connected to it can not be treated equivalently.

Figure 2.7 depicts how blog posts can be modeled using graphs. Edges in the graph represent the relationships between the entities.

![Figure 2.7: Modeling blog posts in a graph database using a directed property graph.](image)

There are two requirements that must be satisfied for a database to be considered a real graph database:

**Index-free adjacency** Each node must have a reference to the nodes that it is connected to \cite{5}.

**Native graph storage** The underlying storage must be designed specifically for managing graphs \cite{5}.

There are graph databases that do not fulfill these two requirements, FlockDB \cite{36} is an example of such a database. Neo4J is a database that satisfies both of these requirements \cite{5} pp.5-7], and is an example of a real graph database.

There are many different graph data models e.g. property graphs, hyper graphs, and triples; the property model being the most popular one in the database domain \cite{5}, p.196].

**The property graph model** This model is represented by a directed graph, it is called a property graph model because both nodes and edges can contain properties. Moreover, each edge can only have two end points in this model, Figure 2.7 shows a property graph.

**Hypergraphs** Unlike property graphs, each edge in a hyper graph can connect more than two nodes, and therefore edges in this model are often referred to as hyper-edges. Although hypergraphs can easily represent many-to-many relationships, such
relationships can also be represented in a property graph. Since it is possible to switch from property graph to hyper graph and vice versa, these two models are isomorphic. Figure 2.8 illustrates a simple hypergraph. In the graph there is a blog post with multiple authors.

**Triples** A triple is a data structure that consists of three parts: subjects, predicates and objects. In Figure 2.8 there is an edge from user 1 to the post and in a triple store this edge would have been stored as “User 1 authored a post”, where “User 1” is the subject, “authored” is the predicate and “post” is the object. A triple store contains a collection of triples, and since triples are independent of each other, it is not possible to perform a rapid graph traversal in this model.

The use cases for the property graph- and hypergraph models are very much the same, as they are isomorphic. The only real difference is that many-to-many relationships are simpler to manage in the hypergraph, which makes databases built on this model slightly more suited towards data models which contain large amounts of such relationships.

The difference between triples and the other two models is that triples only describes relationships. For example, there is no way to save a node that represents a user with multiple properties, such as age, number etc. To accomplish this using triples, one triple is needed for each property: “Bob is 35”, “Bob’s number is 1234”. 

![Figure 2.8: A blog post with three authors, represented using a directed hyper graph.](image-url)
Chapter 3
Axis’ storage solution

In this chapter we will present Axis’ storage solution which is used in their surveillance camera products. First an overview of the data model used to store recordings information is presented, then this model is discussed and evaluated.

As part of the process to evaluate alternative database solutions for storing the recordings information found in Axis’ storage solution, it is first necessary to have a good understanding of how their current storage solution works. This was accomplished by reviewing the available documentation and by exploring the database contents of a running camera.

Axis’ storage solution uses a SQLite database, which is deployed as an embedded C library inside of their camera firmware. SQLite is a very popular database technology that provides a lightweight, zero-configuration, implementation of a relational database with full support for ACID transactions and the SQL querying language.

3.1 Entity-relationship model

To get an overview of how Axis stores recordings information, the tables from a database found on one of their camera products were extracted. As all Axis’ cameras use the same database, the model of the camera did not affect the resulting ER model, which can be found in Figure 3.1 Overall, this model contains most of the information covered in Section 2.2 with the exception of locality and metadata. The lack of metadata may seem confusing as there is an entity called metadata, but this entity is simply a placeholder for future extensions to the database model.

3.1.1 Entities

The primary entity of the model is the recording entity, identified by its primary key id. This entity represents a whole recording, but does not contain the actual footage. Each recording consists of one or more blocks, which represent parts of the captured footage. The relationship between blocks and recordings are managed using foreign keys inside the block entity, which references the primary key of the recording.
The path attribute in the block entity points to the location of the footage, stored using the Matroska Multimedia Container (MKV) standard. The presence of the path attribute means that physically, the footage is stored outside of the database. Thus, the database is only an abstract representation of what footage the camera has captured. This has the benefit of reducing database size, and also makes the footage more manageable. MKV files can simply be copied to a PC and played in any supporting media player, as opposed to requiring the use of software to access the footage inside of the database.

The audios and videos entities provide information about audio and video quality, and the recording sources, events, actions and types entities provide information about how recordings were created. Sources specify which camera captured the recording, events specify which event triggered the recording and actions specify which action the camera took. For example, a “motion detection” event could trigger the camera to record 10 seconds of footage. All of the relationships between recordings and these entities are represented by foreign keys inside the recording entity, referencing the primary keys of the tables.
3.1.2 Model properties

Thanks to the foreign keys used in the recording and block entities, and the separation of audio and video information etc., the model is normalized and counters both redundancy and anomalies.

One drawback which can be identified in the model is the use of numerical identifiers for the primary keys, found in all tables. The purpose of the primary key is to uniquely identify an entity but the numerical values mean that additional database operations are required to access relevant information, such as the name of the event that triggered a recording. For example, a recording triggered by a motion detection event may reference an event with the primary key “1” in the events table. To access the event name, one must first retrieve the identifier from the recordings table and then use this to retrieve the tuple from the events table.

3.2 Accessing recordings information

An important aspect of database design is the knowledge of how the information in the ER model is used. Therefore, it is important to explore what information is accessed from the database, as well as when and how this is done.

Access to the information inside the database is managed by Axis’ camera API, named VAPIX [41]. VAPIX provides an interface which can be used to issue commands and retrieve information from the database. For example, it could be used to retrieve a specific recording using a filename. For the current storage solution VAPIX includes four primary categories of operations:

- Inserting recordings
- Updating recordings information
- Retrieving recordings information
- Removing recordings

As footage is not stored inside the database, creating and updating recordings are simple tasks that only require the insertion of alphanumeric information. The only available customization of the operation is whether the recording should be stored on the camera, or on another network computer.

When retrieving recordings it is possible to apply different filters that affect the set of recordings retrieved from the database. The parameters that can be used to filter recordings are the filename of the recording, its start- and stop time and which event and source that triggered/captured the recording.

Removing recordings is also a simple task as it only requires the filename of any recording that should be removed. Although it involves several operations across multiple tables as block, source, event, action and type records are also removed in the process.

Finally, both retrieving and removing recordings can be performed on all records in the database. A summary of the available operations are given in Table 3.1.

These operations provide the functionality of the database from a user perspective, but it lacks information about how the database operations are performed. To find this information it is necessary to look at each operation and note the queries sent to the database.
Table 3.1: Database operations for the current storage solution.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start a recording</td>
<td>Storage location</td>
</tr>
<tr>
<td>Stop a recording</td>
<td>Filename</td>
</tr>
<tr>
<td>List recordings</td>
<td>Filename, event, source, start- and stop time</td>
</tr>
<tr>
<td>List events</td>
<td>Filename, event, source, start- and stop time</td>
</tr>
<tr>
<td>List sources</td>
<td>Filename, event, source, start- and stop time</td>
</tr>
<tr>
<td>Remove a recording</td>
<td>Filename</td>
</tr>
<tr>
<td>Recording playback</td>
<td>Start time</td>
</tr>
</tbody>
</table>

3.2.1 Inserting and updating recordings

In the current storage solution, inserting a recording involves three steps:

1. If there are no database records for the audio, video, source, event, action and type information of the recording, insert to the corresponding tables. The audio and video information is extracted from the camera’s current settings.

2. Insert the recording and retrieve primary keys from the records created in step 1 to use as foreign keys.

3. Insert a new block and retrieve the filename of the recording created in step 2 to use as a foreign key.

In the first step each table is inspected to see if an entry with the same value already exists. The reasoning behind this is so that multiple recordings can reference the same database entry for audio, video, source, event, action and type information. By performing this step it is guaranteed that all records in these tables are unique and can be used to access all recordings which references them. The second and third steps involve retrieving records from all tables except the recordings table so that foreign keys can be established. In the worst case this process requires nine SQL \texttt{INSERT} operations and seven SQL \texttt{SELECT} operations to insert new records and establish relationships.

Updating recordings

Updating a recording only occurs in two scenarios: when stopping a recording, and when removing blocks. In both of these cases modifications are limited to the start- and stop times of recording and block records. These are the only times SQL \texttt{UPDATE} operations are performed in the current storage solution.

3.2.2 Retrieving recordings, events and sources

Every time a recording is retrieved the query found in Listing 3.1 is used. The query retrieves all the fields from a recording together with the information found in the audios, videos, sources, events, actions and types tables. This is accomplished using the SQL \texttt{LEFT JOIN} operation and the foreign keys described earlier.

Joins are SQL operations that merge the content of two or more tables on some attribute [42]. For example, joining the recordings- and events tables on the primary/foreign key of the event will produce a result where all fields from both tables are included.
3.2 Accessing recordings information

Listing 3.1: SQL query for selecting recordings.

```sql
SELECT * FROM recordings
LEFT JOIN audios ON audios.id=recordings.audio_id
LEFT JOIN videos ON videos.id=recordings.video_id
LEFT JOIN metadata ON metadata.id=recordings.metadata_id
LEFT JOIN recording_types ON recording_types.id=recordings.recording_type_id
LEFT JOIN recording_sources ON recording_sources.id=recordings.recording_source_id
LEFT JOIN recording_actions ON recording_actions.id=recordings.recording_action_id
LEFT JOIN recording_events ON recording_events.id=recordings.recording_event_id
ORDER BY recordings.starttime DESC;
```

Applying filters when retrieving recordings

Applying filters when listing recordings is done by appending different criteria at the end of the query. For example, a single recording can be retrieved by appending the following SQL code which ignores all recordings that do not have the specified filename:

```sql
WHERE recordings.filename='20150224_114643_3C50_00408CC5A111';
```

Table 3.1 shows that several different parameters could be used to filter recordings. All of these will result in queries similar to the one above. Another example is when the recordings in the result should be filtered on both their start- and stop time:

```sql
WHERE recordings.starttime < '2015-03-08T16:13:05.609173Z' AND
  recordings.stoptime > '2015-03-08T14:13:05.609173Z';
```

3.2.3 Removing recordings

In Axis’ storage solution removing a recording involves three steps:

1. Remove the recording using the provided filename.
2. Find all blocks that belong to the recording, using the filename, and remove them.
3. Check whether any records in the audios, videos, sources, events, actions and types tables are referenced by a recording. If a record is found to be unreferenced it is also removed.

The first and second steps remove the recording and blocks from the database. The third step is performed to keep the database clean of unused records and to avoid unnecessary comparisons against unreferenced records. These may occur when searching for a value in a table as each entity is checked using the condition. Performing step 3 includes one logical \((\text{NOT})\) operation, as well as a \(\text{SELECT}\) operation, to remove entities from the tables. This process is performed using \(\text{nested}\) queries with the following structure:

```sql
DELETE FROM audios WHERE id NOT IN (SELECT DISTINCT audio_id FROM recordings);
```
3.2.4 Recording playback

Recording playback involves streaming footage from the camera to the users browser and consists of two steps: First recordings are retrieved using the query in Listing 3.1 so that the user can get some basic information about the recording he/she is viewing. Then blocks are retrieved so that the footage can be accessed using the path field. Recordings are retrieved in the same way as earlier by filtering using the filename (WHERE recordings.filename = '...'), and block information is retrieved by selecting two hours worth of blocks starting from the specified start time.

3.3 Pros and cons

The ER model presented in Figure 3.1 is not very complex, as there are only one-to-many relationships. There is also only one entity with multiple relationships (the recordings entity). However, there are both pros (+) and cons (−) with the ER model connected to how the model is used:

+ The model contains few relationships. This means that the ER model is easy to extend and maintain, as key-constraints are easy to manage.

+ The model is normalized. By normalizing, both redundancy and anomalies are deleted from the model, which makes maintenance easy.

− Normalization and numeric primary keys results in the need for joins to access all recordings information. As joins are expensive operations, the normalization and numeric primary keys directly impacts the performance of the database.

− Normalization requires multiple table-lookups in order to keep the database “clean” after removing recordings. Checking for unreferenced entities in the tables results in overall lower performance. Also, there is minimal impact on the maintenance of the database, as unreferenced entities do not require any maintenance.
In this chapter we will evaluate which database paradigms could be used to represent the recordings information found in Axis’ data model. The selection of paradigms to evaluate will be based on a list of candidate database systems which are selected after applying some requirements defined by Axis.

4.1 Requirements

Axis’ cameras are embedded systems which rely on limited amounts of resources and strict software requirements. Because of this, there are some limitations to what software can be used on their cameras. Axis defines the following requirements which should be applied when evaluating the potential use of new database technologies on their cameras.

Open source It is important that the source code for the database implementation is available so it can be properly cross-compiled for use on an Axis camera. Open source software is also often accompanied by a community of developers who continuously evaluate the software from different perspectives, such as security and stability.

Licensing To make modifications possible the database source code must be distributed freely, and this requires the use of an open source license. There are several different variants of such licenses available, but as Axis distributes proprietary software not just any license can be used. Thus, any license that includes a strict copyleft notice that would force Axis to open source their firmware cannot be used. Using an open source license is also important to avoid licensing fees.

Linux support At the lowest level of the camera firmware, there is a Linux kernel which manages system resources, services etc. Therefore, it must be possible to compile the database for the Linux operating system.

Implementation language Axis currently only supports code written in C and C++ which can be cross-compiled for their cameras. This means that the database must either be implemented in one of these two languages, and use libraries available on
the camera system (which can be cross-compiled themselves), or provide a database binding to the server (in the case of a client/server system) for either language.

**CPU and memory usage** Even though the camera can be considered a small computer, it still has very limited processing power and memory. Thus, the database should not consume more resources than the current solution. If it does, it should be possible to motivate the additional resource consumption.

As server-based databases have to share resources with the rest of the system, it is not feasible for a database server to consume the majority of the camera’s resources.

**Small file size** Axis’ cameras store their recordings on an SD card or a NAS and the majority of the available disk space should be used to store footage and not database files. A sample SQLite database with 10000 “dummy” recordings (a recording with one block that does not actually represent any footage on the SD card) consumes 5MB of disk space. An alternative solution should not be significantly larger than this when containing the same amount of information.

**Small installation size** Axis’ cameras come in many different configurations, but as an embedded system most cameras are only provided with enough internal storage to support the camera firmware and default applications. Some databases might need binary files and other dependencies to be installed onto the camera. The size of these binaries, as well as their dependencies, have to be small so that they fit on the camera. The specific camera that is used in this thesis (see Table 6.1, Chapter 6) is limited to 128MB of internal storage, with only 50MB available for binary files and dependencies with the firmware and default applications installed.

### 4.2 Survey of NoSQL databases

In an initial effort to find a database to evaluate against the SQLite database in the current storage solution, we performed a survey for suitable candidates. In the survey, online resources measuring database popularity [27] were used to produce the list of candidates in Table 4.1.

Amongst the candidates, three paradigms are found to be missing, namely the NewSQL, column and object-oriented paradigms. The reasoning behind NewSQL and column-oriented being left out is that both are based on the relational paradigm. As SQLite is also based on this paradigm, being a RDBMS, we chose to look closer at other paradigms and see what they have to offer. Column-oriented databases focus on column-based operations (see Chapter 2) which the ER model does not rely on (see Section 3.2). The object-oriented paradigm is also out as it aims to provide persistence to object-oriented programming languages. However, the current storage solution is implemented in C which is not an object-oriented language.

### 4.2.1 Database selection based on the requirements

In the following section the requirements from Section 4.1 are applied to the candidate systems in Table 4.1 to dismiss the unsuitable databases (and subsequently their paradigms).
### 4.2 Survey of NoSQL databases

#### Table 4.1: Collection of database candidates.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>License</th>
<th>Language</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aerospike</td>
<td>Apache v2.0</td>
<td>C</td>
<td>Client/Server</td>
</tr>
<tr>
<td>2</td>
<td>BangDB</td>
<td>BSD</td>
<td>C/C++</td>
<td>Library</td>
</tr>
<tr>
<td>3</td>
<td>BerkleyDB</td>
<td>AGPL3</td>
<td>C</td>
<td>Library</td>
</tr>
<tr>
<td>4</td>
<td>HamsterDB</td>
<td>Apache v2.0</td>
<td>C/C++</td>
<td>Library</td>
</tr>
<tr>
<td>5</td>
<td>LevelDB</td>
<td>BSD</td>
<td>C++</td>
<td>Library</td>
</tr>
<tr>
<td>6</td>
<td>Redis</td>
<td>BSD</td>
<td>C</td>
<td>Client/Server</td>
</tr>
<tr>
<td>7</td>
<td>Riak</td>
<td>Apache v2.0</td>
<td>C, Erlang</td>
<td>Client/Server</td>
</tr>
<tr>
<td>8</td>
<td>RocksDB</td>
<td>BSD</td>
<td>C++</td>
<td>Library</td>
</tr>
<tr>
<td>9</td>
<td>Tokyo Cabinet</td>
<td>LGPL</td>
<td>C</td>
<td>Library</td>
</tr>
<tr>
<td>10</td>
<td>Voldemort</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Client/Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Accumulo</td>
<td>Apache v2.0</td>
<td>Java, C++</td>
<td>Client/Server</td>
</tr>
<tr>
<td>12</td>
<td>Cassandra</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Client/Server</td>
</tr>
<tr>
<td>13</td>
<td>HBase</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Client/Server</td>
</tr>
<tr>
<td>14</td>
<td>Hypertable</td>
<td>GPL3</td>
<td>C++</td>
<td>Client/Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>BaseX</td>
<td>BSD</td>
<td>Java</td>
<td>Client/Server</td>
</tr>
<tr>
<td>16</td>
<td>Couchbase Mobile</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Library</td>
</tr>
<tr>
<td>17</td>
<td>CouchDB</td>
<td>Apache v2.0</td>
<td>Erlang</td>
<td>Client/Server</td>
</tr>
<tr>
<td>18</td>
<td>EJDB</td>
<td>LGPL</td>
<td>C</td>
<td>Library</td>
</tr>
<tr>
<td>19</td>
<td>JasDB</td>
<td>MIT</td>
<td>Java</td>
<td>Library</td>
</tr>
<tr>
<td>20</td>
<td>MongoDB</td>
<td>Apache v2.0</td>
<td>C++</td>
<td>Client/Server</td>
</tr>
<tr>
<td>21</td>
<td>UnQLite</td>
<td>BSD</td>
<td>C</td>
<td>Library</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Bitsy</td>
<td>AGPL3</td>
<td>Java</td>
<td>Library</td>
</tr>
<tr>
<td>23</td>
<td>FlockDB</td>
<td>Apache v2.0</td>
<td>Scala</td>
<td>Client/Server</td>
</tr>
<tr>
<td>24</td>
<td>HyperGraphDB</td>
<td>LGPL</td>
<td>Java</td>
<td>Client/Server &amp; library</td>
</tr>
<tr>
<td>25</td>
<td>Neo4j</td>
<td>GPL3</td>
<td>Java</td>
<td>Client/Server &amp; library</td>
</tr>
<tr>
<td>26</td>
<td>Titan</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Client/Server</td>
</tr>
<tr>
<td>27</td>
<td>Weaver</td>
<td>BSD</td>
<td>C, Python</td>
<td>Client/Server</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>ArangoDB</td>
<td>Apache v2.0</td>
<td>C/C++</td>
<td>Client/Server</td>
</tr>
<tr>
<td>29</td>
<td>OrientDB</td>
<td>Apache v2.0</td>
<td>Java</td>
<td>Client/Server &amp; library</td>
</tr>
<tr>
<td>30</td>
<td>WhiteDB</td>
<td>GPL3</td>
<td>C</td>
<td>Library</td>
</tr>
</tbody>
</table>
Open source and Linux support

All databases except BangDB \cite{45} are considered open source\footnote{The BangDB source code is currently not available for download. Upon asking, the developer has stated that the source should become available in the fourth quarter of 2015.}, as their source code is available for free on their respective websites. All of these also support compilation on Linux or provide distribution-specific packages for their databases.

Implementation language and installation size

As Axis only supports the cross-compilation of C and C++ code, databases implemented in languages such as Erlang or Python can be dismissed. Databases written in Java, being a cross-platform language, are also dismissed as integrating Java with C and C++ requires the use of the Java Native Interface (JNI). JNI is a framework that enables native applications to communicate with programs running on a Java Virtual Machine (JVM), but the process of using JNI is quite complex and there are dangers associated with memory management \cite{67}.

Another problem with Java, as well as Scala, is that they require the runtime system to be installed and this requires more than the 50MB of available disk space. There are special embedded versions of Java available \cite{68,69}, but it would require the cross-compilation of the JVM and other components. Since there are several candidates available with implementation in C/C++, databases implemented in Java or Scala were not considered good candidates.

Size is also an issue for some of the more popular databases amongst the candidates that are aimed towards distributed- and web applications (Accumulo \cite{53}, Cassandra \cite{9}, HBase \cite{54}, Hypertable \cite{55}, CouchDB \cite{57}, MongoDB \cite{7}, HyperGraphDB \cite{62}, Neo4j \cite{6}, Titan \cite{63} and ArangoDB \cite{65}).

Licensing

When it comes to licensing all databases which enforce strict copyleft have to be avoided. Only one of all databases not written in Java, Scala, Erlang or Python enforces such licensing agreements, namely BerkleyDB \cite{70}.

Weaver, while not enforcing copyleft, must also be dismissed as the developers explicitly state that their database may not be used in proprietary software at this point in time \cite{64}.

Resource consumption

For resource consumption all databases that rely on a server to host the database will consume more resources than those providing embedded library solutions. This is attributed to the fact that an additional server process has to be run on the camera and even if this process was mostly idle it would still consume some amount of memory and CPU cycles. The latency introduced when communicating with the server is also another factor not found in embedded libraries that affects overall performance. As there are a good number of both server-based and embedded library solutions, the choice is made to dismiss server-based solutions based on these factors.

4.2.2 Updated list of databases

After going through the requirements, a new collection of candidate database systems can be established. This new list can be found in Table 4.2 and contains only server-less databases, implemented in C or C++. The list contains databases from two paradigms.
only, namely key-value and document. The two paradigms provide different storage solutions and must be further evaluated to see which one is more preferable when used to model the current storage solution.

Table 4.2: Final selection of databases and paradigms to be considered for evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>HamsterDB</td>
<td>C/C++</td>
<td>Key-value</td>
</tr>
<tr>
<td>LevelDB</td>
<td>C++</td>
<td>Key-value</td>
</tr>
<tr>
<td>RocksDB</td>
<td>C++</td>
<td>Key-value</td>
</tr>
<tr>
<td>Tokyo Cabinet</td>
<td>C++</td>
<td>Key-value</td>
</tr>
<tr>
<td>EJDB</td>
<td>C</td>
<td>Document</td>
</tr>
<tr>
<td>UnQLite</td>
<td>C</td>
<td>Document</td>
</tr>
<tr>
<td>WhiteDB</td>
<td>C</td>
<td>N-tuple (document)</td>
</tr>
</tbody>
</table>

4.3 Recordings information in a key-value database

The basic implementation of the key-value database only provides pairs of keys and values where values can be retrieved using a key (see Section 2.4.1). This means that values themselves cannot be used in any logical operations, as the database does not know about their contents. Such operations would have to be performed by first retrieving values using keys and then by performing the operation on these. As there are several instances where logical operations are needed in the ER model, like when filtering results (see Section 3.2), the logical operations must be implemented by a developer inside the application (or in the application layer). Depending on the programming skills of the developer this does not necessarily introduce much overhead compared to a database that supports value-based queries, as it is very fast to retrieve values ($O(1)$ time [71, p.372]). However, implementing the logical operations does introduce more complexity into the application layer.

The question is then how the entities in the ER model (see Figure 3.1) should be modeled using keys and values. It is possible to extract each field in every entity as its own key-value pair, but as recordings are always retrieved together with all of their fields, this would only incur extra database operations. Instead, it is more reasonable to keep the entities as they are but store all fields of the entity as a single value (see Listing 4.1). It would then be possible to extract a recording with only a single database operation, and the value of the recording could be tokenized to access the individual fields.

Listing 4.1: Merging all fields for a recording with the key H3FA4EB

```plaintext
'H3FA4EB': {'filename': 20121206_181501_A9CD_00408CC5A88D,  
            'path': 20121206/18,  
            'starttime': 2012-12-06T18:15:01.650780Z,  
            'stoptime': 2012-12-06T18:15:02.773841Z,  
            '...'}
```

Using keys to organize entities

When storing entities using the format described in Listing 4.1, it becomes hard to access specific recordings. For example, how can the recording in Listing 4.1 be retrieved when
one only knows the filename? As the filename is contained inside the value it cannot be used for comparisons until the value has been retrieved and tokenized.

One solution is to establish a pattern for keys which can be used to reference records. For example, by establishing that a recording should have the following key: '<filename>', retrieving recordings is trivialized. This pattern could then be extended to include blocks: '<filename>:<block_id>'.

**Optimizing functionality by regrouping entities**

When keeping all of the entities in the original ER model and translating them into key-value pairs, additional complexity on the application layer is necessary to perform logical operations and the aggregation of multiple tables (previously performed using joins or select statements, see Section 3.2).

When keeping all of the entities in the original ER model, translating them into key-value pairs, additional complexity on the application layer is necessary to perform logical operations, and the aggregation of multiple tables (previously performed using joins or select statements, see Section 3.2).

While most logical operations cannot be avoided, it is possible to avoid the need for joins by "denormalizing" the ER model. This is accomplished by taking the information previously retrieved using joins and embedding it inside the recording entity, replacing the foreign keys with actual values. The process of denormalizing the model indicates that normalization is lost. When merging the entities as described above this is exactly what happens, as both redundancy and update anomalies are introduced (data is duplicated across values). Note that deletion anomalies are not introduced, as one actually wants information to be lost when removing recordings (see Section 3.2.3 where tables are "cleaned" after removing recordings).

Additionally, if one considers the operations described in Section 3.2 it is noted that the only information which is updated in the ER model are the start and stop times of recordings and blocks. This means that update anomalies will never occur, even if they theoretically could.

By denormalizing the data a recording, complete with audio, video etc. information, can be retrieved using a single query in $O(1)$ time.

Denormalization introduces another problem, namely when querying recordings based on events and sources. In both of these cases all recordings would have to be retrieved, tokenized and then compared in order to filter on these fields. One way to avoid retrieving all recordings when searching using event name is to save a list of all recordings that reference an event entity, and then access this list using the event name. This way of efficiently accessing records by means of some piece of information other than the primary key (or in this case, just the key) is called a *secondary index* [72]. For events and sources the following key-value pairs can be used to build such indices:

''event_name'' : {<filename>, <filename>, ...}
''source_name'' : {<filename>, <filename>, ...}

Additionally, a secondary index enables one to retrieve the identifiers for all blocks in a recording: '<filename>': {<block_id>, <block_id>, ...}.

**4.3.1 Pros and cons**

The key-value paradigm introduces some new ways to manage recordings information compared to the relational paradigm, and provides an alternative model for storing the
4.4 Recordings information in a document database

A document can be viewed as a combination of a relational database table and a key-value store, where each tuple is represented as a collection of key-value pairs (see Chapter 2).

One major difference that distinguishes the document database from the key-value database is that most document databases support value based queries using the query language of the database or map/reduce functions [7, 8, 57–59]. With this functionality there is no need for complexity in the application layer as logical operations can be performed as a part of the query (similar to the SQL SELECT WHERE statement).

Organizing entities

As each document provides its own internal key-value store, it is possible to represent an entity from the ER model (see Figure 3.1) by turning each field into a key-value pair. For the key-value database this was not very useful as recordings were retrieved together with all of their fields, but as documents are retrieved as a whole (see Section 2.4.2) this is not an issue. It is then possible to simply translate the current ER model into a document database by recreating each entity and use the same fields and values. For example, a recording could be modeled using the following document:

document: recording {
    '_id': 'H3FA4EB',
    'filename': '20121206_181501_A9CD_00408CC5A88D',
    'path': '20121206/18',
}
'starttime': '2012-12-06T18:15:01.650780Z',
'stoptime': '2012-12-06T18:15:02.773841Z',
...

In the key-value store it was suggested that these entities could be organized by using key patterns. In the document database these patterns are not necessary as values can be compared against each other as a part of the query. To retrieve the above document, without knowing the id, it would be possible to construct a query that selects the recording with the filename ‘20121206_181501_A9CD_00408CC5A88D’.

Optimizing functionality by regrouping entities

In the key-value model it was suggested to embed information to avoid the need for aggregations (joins and extra select statements). While value based operations are supported in a document database, most databases lack support for operations equivalent to the SQL join [73, 74]. Performing these operations would require work on the application layer, as multiple documents must be retrieved and combined.

Denormalization by embedding information directly into the document avoids work on the application layer, and secondary indices can be used to facilitate access to all recordings given an event name and/or a source name. However, denormalizing does introduce both redundancy and anomalies (see Section 4.3).

Additionally, secondary indices are not necessary in the document database as you can query on values. For example, all blocks in a recording could be retrieved by comparing the foreign keys inside all blocks (see the ER model, Figure 3.1). In this scenario secondary indices provide some convenience, as there is no need to retrieve all blocks (only the ones referenced in the secondary index). An illustration of these two methods of referencing blocks is depicted in Figure 4.1.

**Figure 4.1:** Establishing relationships between block- and recording documents using references (top) and secondary indices (bottom).
4.4.1 Pros and cons

The document model provides much of the same capabilities as the key-value model, and this is reflected in the list of pros (+) and cons (–) below.

+ The concept of documents that contain fields and values is familiar to those with knowledge about relational databases. While this does not impact performance or maintenance, it is generally positive that the model is understandable.

+ Being able to query for individual fields means less complexity in the application layer, compared to a key-value database. This makes the database more easy to use and reduces maintenance time (as one only has to overlook the queries themselves, and not the tokenization).

+ A schema-less model makes denormalizing fields a natural choice, allowing joins to be completely avoided. As joins are expensive operations, there are additional performance to be gained when avoiding them. It does however introduce some redundancy and anomalies. Fortunately, the anomalies are no problems for the current model.

+ Secondary indices increase performance by making it possible to quickly access key-value pairs based on fields instead of keys.

– Secondary indices introduce additional maintenance as they have to be updated and reviewed each time records are added or removed.

4.5 Selecting a paradigm

Overall, the document database provides the same features as the key-value store, but with the additional structure that organizing key-value pairs allows for. It is also possible to perform value based queries and logical operations in the database instead of the application layer.

The fact that value based queries have to be implemented on the application layer for the key-value model does not necessarily mean a degradation in performance, due to the $O(1)$ lookup times, but it does mean that the key-value model will be more complex as a result. Working on the application layer also means that alterations to the model require more work as the format of the values in the model affect how they are handled in the application layer, due to tokenization.

In the end, the choice of paradigm will come down to convenience. A document database provides the same flexibility as a key-value database using embedded information and secondary indices. It also supports queries for individual fields and indexes. We find that these are enough reasons to favor a document database over a key-value database.
Chapter 5

Solution

In this chapter we will continue the analysis from the previous chapter and present the database, and data model, that are to be compared against Axis’ SQLite database. First we look at the remaining databases and select one of these based on their features, then a general document ER model is presented. Finally this model is adjusted for the selected database.

5.1 Database candidates

After the analysis three library-based candidates had been identified, namely the Embedded JSON Database Engine (EJDB), WhiteDB and UnQLite. These databases are all based on the document paradigm, but provide very different implementations and features.

5.1.1 UnQLite

UnQLite is a hybrid database that supports both the key-value and the document paradigms. It is implemented as a self-contained C library which has minimal dependencies on external libraries, and is similar to SQLite in that it stores the whole database in a single file. The database was created by Symisc Systems [75] in 2013 and the latest version (1.1.6) was released in August 2013.

From a document perspective the UnQLite storage engine is capable of storing JSON documents. However, it lacks some features commonly associated with the paradigm such as a way to group documents (collections), indices and more importantly, value based queries. The reason for this is that the database implementation is more similar to a key-value store in that one can save key-value pairs and store JSON documents as the values.

UnQLite provides support for proper ACID transactions and the storage engine provides $O(1)$ lookup times thanks to the architecture. It is also flexible as its storage engine can be changed during run-time. There are currently two different storage engines available: one where the database is stored in-memory as a hash table, and one where it is stored persistently on the file system [60].
5.1.2 WhiteDB

WhiteDB is an in-memory database, implemented as a standalone C library. It stores N-tuples by reading and writing data directly to the main memory of the host system. The database was released in 2008 by the WhiteDB Team \[66\] and the latest version (0.7.3) was released in November 2014.

An N-tuple stored in WhiteDB can be viewed as a more structured version of a key-value pair, where a key can be used to access a value. These values, however, are not stored as single entities, but are instead split into lists where each element is accessible. This makes value based queries possible.

WhiteDB stands apart from other document databases in that it can only store values in a numerical format, called WhiteDB Ints \[66\]. This means that all alphanumerical values need to be encoded before being stored in WhiteDB. The internal architecture is based on the network mode, which is very similar to the graph model, and this makes it possible to link records to each other and create graph structures. This feature makes joins unnecessary in WhiteDB since it is possible to place a direct reference to records instead of storing their identifiers.

In WhiteDB’s design there are no locks applied to any part of the database during runtime. This can have a negative impact on the data stored in the database if two or more processes start modifying the content simultaneously, as the database could end up in an inconsistent state. WhiteDB does provide options to enable database locks temporarily, but this manual concurrency control forces the application to perform this task.

5.1.3 EJDB

EJDB is an embedded document database implemented as a standalone C library. Its implementation is based on the core of the Tokyo Cabinet key-value storage engine, and its design is highly inspired by the more popular MongoDB document database. As such EJDB has support for many of the same features as MongoDB such as indexing, value based queries, collections and an expressive querying language \[58\]. The database was created by Softmotions \[76\] in 2012, the year it was officially open sourced, and is actively maintained by the author. The latest version (1.2.7) was released in April 2015.

In EJDB documents are stored using the JSON format, transformed into a binary representation called Binary JSON (BSON). Documents are managed using collections which are used to group documents with similar purpose together. Each collection is stored as a separate file on the file system, accompanied by a “master” database file which links all collections together. This model makes it possible to reuse collections in different databases, as long as the collection is first created in the master database file and then overwritten. Each collection has a minimal file size of 532.5kB.

EJDB also provides a limited implementation of ACID transactions which provide atomic, durable non-parallel and read-uncommited collection level transactions \[58\].

5.1.4 Selecting the database

Overall, both WhiteDB and EJDB are more preferable than UnQLite as they provide more document-related features. UnQLite is not a good candidate because it lacks many of the feature described in Section 5.1, such as indices, collections and value based queries.

WhiteDB appears to be a good candidate, as it provides indices and value based queries. Although it lacks collections, with indices the grouping of records is not really
necessary. It also provides persistent storage as a complement to the in-memory architecture, though compared to EJDB it requires more work to use since the structure has to be built using N-tuples and the network model. Documents also have to be encoded before they can be stored.

These limitations for UnQLite and WhiteDB mean that EJDB will be our choice for further comparison against Axis’ SQLite database.

5.2 Document data model

Based on the discussion in the previous chapter the ER model in Figure 5.1 was created for EJDB. This model includes three collections (recordings, blocks and events & sources) and four different document types (recording, block, sources and events). The model is fairly similar to the one found in Figure 3.1 and is centered around the recording document. In this document the foreign keys have been replaced by embedding tuples from the audios, videos, sources, events, actions and types tables in Figure 3.1. The model also includes secondary indices to help with the retrieval of recordings based on event- and source names.

Figure 5.1: ER model for the EJDB document database.
5.3 Adjusting the model for EJDB

In the following sections we will discuss some design choices that were made when creating the ER model in Figure 5.1. Before a model could be used in EJDB it had to be evaluated and adjusted so that there were no problems with the model. This includes checking so that all needed functionality is supported (see Section 3.2), and that the performance is acceptable.

5.3.1 Indices

EJDB provides indices implemented as B+ trees which can speed up the execution time for value based queries [77]. In Section 3.2 four values were used to perform value based queries, namely event and source name as well as start and stop times. To see if any of these values could prove to be good candidates for indices, we performed two small experiments. These experiments were performed using a database that contained 10000 recordings and each recording had one block. There were four different event names, each with 2500 recordings.

First the event names were indexed, and all recordings were retrieved using the event name “ManualTrigger1”. Secondly, start- and stop times were indexed, and recordings selected by filtering using both of these times. The results of these experiments can be found in Table 5.1.

<table>
<thead>
<tr>
<th>Table 5.1: Average execution times (s) when selecting and filtering recordings with indices.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing on event name</td>
</tr>
<tr>
<td>With index</td>
</tr>
<tr>
<td>0.002041</td>
</tr>
</tbody>
</table>

From these experiments we noticed that an index on the event names reduced the time it took to retrieve recordings by 30%. However, an index on start- and stop times did not provide any increased performance. We concluded that indices only provide increased performance when added on fields used in equality operations. Thus, our document ER model will have indices on both event- and source names in the recordings collection.

5.3.2 Secondary indices

In Section 4.3 secondary indices were mentioned as a method to access records by means of some piece of information other than the primary key.

One problem that we noticed, when initially creating collections for the secondary indices, was that these grew rapidly in size. In a database with 10000 recordings, each with one block and four events, each referenced by 2500 recordings, the events collection had a size of 32.5MB. Since the recordings collection (with 10000 records) had a size of 6.9MB this required further investigation.

To check whether a larger size could be motivated by increased performance, we performed a small experiment where all recordings which referenced a specific event were retrieved. The experiment was performed on a computer with the hardware configuration found in Table 5.2 and yielded the results in Table 5.3. From these times we concluded that the secondary indices did not provide much benefit over simply querying the recordings document for the event name.
5.3 Adjusting the model for EJDB

Table 5.2: Hardware configuration for the host PC used in some local experiments.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7 860 @2.8GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>16GB RAM</td>
</tr>
<tr>
<td>Hard drive</td>
<td>240GB Intel SSD</td>
</tr>
</tbody>
</table>

Table 5.3: Average execution times (s) when selecting recordings using a secondary index on event name.

<table>
<thead>
<tr>
<th></th>
<th>With secondary index</th>
<th>Without secondary index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.023548</td>
<td>0.006772</td>
</tr>
</tbody>
</table>

Even though the secondary indices were not useful for querying recordings using source and event names, they were still useful to retrieve lists of all sources and events in the database. As such the secondary indices were repurposed into the following format where they list the names, as well as how many times they are referenced:

collection:events & sources {
  document:event {
    '_id': '1',
    'motion detection': 2,
    ...
  }
  document:source {
    '_id': '1',
    'camera 1': 2,
    ...
  }
}

5.3.3 Querying start times

EJDB has not created its own querying language, but instead mimics the querying language specified in the MongoDB documentation [78]. This language allows the specification of fields, values and conditions which are evaluated to perform the requested database operation.

In EJDB queries are created as objects where you append different fields, values and conditions. An example of this can be found in Listing 5.1 where a single recording is retrieved using the following “Mongo-style” query:

```json
{ 'filename': '20150224_114643_3C50_00408CC5A111' }
```

For more information on how more complex queries are created in EJDB the reader is directed towards the EJDB documentation and source code [79, 80].

Listing 5.1: Construction of a EJDB query to retrieve a single recording.

```c
bson bq1;
bson_init_as_query(&bq1);
bson_append_start_object(&bq1, "filename");
bson_append_string(&bq1, "20150224_114643_3C50_00408CC5A111");
bson_append_finish_object(&bq1);
bson_finish(&bq1);
```
One limitation with EJDB queries are that string comparisons only support the equality operation (=). In the current storage solution start- and stop times are only stored as strings, and are compared against each other using conditional operators (<, >) (see Section 3.2). To solve this problem we included two additional fields inside the recording document: starttime.c and stoptime.c which are numerical representations of the timestamps. For example, the timestamp 2015-03-08T16:13:05 would be encoded as 20150308161305, or in epoch time (number of seconds that have elapsed since January 1, 1970).

5.3.4 Collection joins

One feature which makes EJDB stand apart from other document databases is that it provides an implementation of the SQL join operation. These joins are limited to collections and are called collection joins. Collection joins make it possible to access information from multiple documents in multiple collections in a single query. Collection joins are limited to document identifiers (see Section 2.4.2), so two collections cannot be joined on specific field values.

As mentioned in Chapter 4, it is possible to create a document model which is identical to the ER model in Figure 3.1, including the same entities and relationships (without key-constraints). This approach relies on references inside a recording to point to the corresponding audios, videos, sources, events, actions and types entities. These can then be used in collection joins to retrieve all information about a recording.

Testing collection joins as an alternative to embedding documents

With two possible ways to query for recordings information, either by embedding or by using collection joins, it is important to evaluate both solutions. In an attempt to do this, we performed an experiment where a database with 10000 recordings was used with two different configurations.

In the first configuration one collection was used, and contained documents which had both audio and video information embedded. Retrieving information from this document does not require collection joins. In the second configuration the audio and video information was separated into their own collections and documents. This configuration requires collection joins to access all information at once.

Recording, video and audio information was then queried from the collections/documents in both configurations, and execution time and database sizes (summation of all files) were measured. The experiments were executed on a PC with the hardware configuration found in Table 5.2 and the execution times and sized are shown in Table 5.4.

From these results we see that the execution time when using collection joins is about 700% longer than when querying recordings with audio and video information embedded. We also see that the database size increases, something which is credited to the fact that each collection has a minimum size of 532.5kB.

<table>
<thead>
<tr>
<th>With collection joins</th>
<th>Without collection joins</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.042584</td>
<td>0.006026</td>
</tr>
<tr>
<td>10.6MB</td>
<td>10.3MB</td>
</tr>
</tbody>
</table>

Table 5.4: Average execution times (s), and database size, when selecting recordings information with collection joins.
When a collection join is performed in EJDB, it is done by comparing document identifiers. This means that EJDB must iterate over all documents in both collections in order to successfully join the collections. If all documents are not covered, there is no guarantee that all document identifier-pairs have been found. These comparisons will take longer time to perform, compared to simply retrieving all documents from a collection. This is likely the reason why collection joins exhibit longer execution times.

Overall, using collection joins instead of embedding information is an inefficient solution.
In this chapter we will introduce the system configuration for the camera, and the test cases which will be used to evaluate the new EJDB database against the baseline SQLite database. First the performance factors and benchmark methodology is motivated, and then test data and test cases are presented.

### 6.1 System configuration

The system on which the benchmark will be performed is a Axis p3367-V camera \[82\]. The relevant hardware information for the camera can be found in Table 6.1. The primary storage is used to install system software components, while the secondary storage is used to store database files.

<table>
<thead>
<tr>
<th>CPU</th>
<th>ARTPEC-4 [83]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size</td>
<td>128MB RAM</td>
</tr>
<tr>
<td>Primary storage</td>
<td>128MB NAND flash</td>
</tr>
<tr>
<td>Secondary storage</td>
<td>16GB Kingston SD Card (Class 10)</td>
</tr>
</tbody>
</table>

### 6.2 Performance factors

To measure the performance of the databases, *performance factors* have to be established. These factors define what properties should be measured in order to gather the results necessary for evaluation of the databases. As all factors do not have the same weight when evaluating the performance for the databases, each factor is given a different priority. The following factors are defined for the P3367-V camera:

- **Prio 1 – Execution time** Average time performing database operations. This is an
6. Evaluation

important measurement as it directly affects user experience. Slow response times means longer waiting periods when interacting with the system.

**Prio 2 – Memory usage** Average memory usage while performing database operations. The memory consumption is also an important measurement, as it affects overall performance. If too much memory is used, paging [84] might occur which would cause the system to slow down.

**Prio 3 – CPU usage** Average CPU utilization while performing database operations. The CPU utilization is important from a system perspective, as a high CPU utilization has a direct impact on the performance of the camera system. A process that consumes 100% of available resources will force the other processes to wait and can result in sluggish behavior.

**Prio 4 – Read/Write operations** Storage media utilization (bandwidth) encountered while writing and reading from the database on an SD card. SD Cards have a limited lifespan [85], so the quantity of data that is read and written has a direct impact on the lifespan of the card.

**Prio 5 – Database size** Physical size of the database.

### 6.3 Benchmarking the databases

There are two different approaches to benchmarking, namely *application-specific* and *generic* benchmarks [86]. In the evaluation of EJDB and SQLite an application-specific benchmark will be used, as it provides specific results for Axis’ storage solution. In this approach test cases are designed based on the functionality of the application, and test data represent real information that can be found in the application.

#### 6.3.1 Test data

In [86] two methods for obtaining a database with test data are presented: either an already existing database is used, or a synthetic database is generated. The first method is a good choice when the database environment is already known as the use of real data provides more applicable results. The latter is more suited towards generic benchmarking, and provides greater control over selectivity and can lead to more precise results [86].

A combination of both, where real test data is generated, will be used in the test cases. A set of 10000 recordings and blocks (each recording has one block) will be generated, together with one audio, video, source, action and type record. In EJDB this information will be embedded inside the recordings document. Four different events will also be generated, and the set of recordings will be divided equally across all events, resulting in 2500 recordings associated with each event. This generated data will also be used for comparing database sizes.

#### 6.3.2 Test cases

Axis have implemented their own benchmarking suite which collects performance results for their storage solution. This benchmark provides measurements for all factors listed earlier. As the benchmarks in the suite are designed specifically for Axis storage solution,
reusing these in our benchmark will result in an application-specific benchmark with good coverage.

Axis’ test cases cover three different categories, namely the insertion, retrieval and removal of recordings. The following list includes all test cases, where test case 1 and 2 are covered in the first category, 3 to 14 in the second and 15 to 17 in the third.

TC1 Insert a new recording
TC2 “Stop” a recording by updating its stop time
TC3 List all recordings in descending order
TC4 List all recordings in ascending order
TC5 List all recordings, limit number of results to 20
TC6 List all recordings, limit number of results to 100
TC7 List all recordings, limit number of results to 1000
TC8 List all recordings, limit number of results to 20, start from result 100
TC9 List all recordings, limit number of results to 100, start from result 1000
TC10 List all recordings, limit number of results to 1000, start from result 5000
TC11 List all recordings based on event name
TC12 List all recordings one hour later than the first recording’s start time
TC13 List all recordings one hour earlier than the last recording’s stop time
TC14 List all recordings one hour later than start time and one hour earlier than stop time of the first and last recordings, respectively
TC15 Remove 50 recordings
TC16 Remove all recordings
TC17 Remove all recordings that started before the first recording’s start time, appended with 10 minutes

Inserting recordings

When inserting a recording in SQLite, the process described in Section 3.2 is used. For EJDB all recordings information, except blocks, is inserted into a single document, and additional event- and source information is inserted into the events- and sources documents. These operations are all performed as part of a transaction in both databases.

Retrieving recordings

When retrieving all the recordings in SQLite, the query in Listing 3.1 is used. This is then followed by either a nested SQL ORDER BY, LIMIT or OFFSET query. Due to the normalization described in Chapter 5 EJDB only retrieves single documents from the recordings collection.

In TC12, TC13, TC14 and TC17, additional database operations are performed to retrieve the start- and stop time from the first and last recording in the database, respectively.
Removing recordings

For SQLite, removing recordings follows the process described in Section 3.2. In EJDB the same effect is achieved by removing a recording document, and then reduce the corresponding count in the sources/events documents.

6.4 Execution times

When measuring execution time it is important that the time measured is attained correctly, and that only time spent performing database operations is included in the measurement. To make sure that the correct measurements are collected, the GNU C Library timeval data type will be used to measure execution times, according to the GNU libc manual [87]. The timeval data and the gettimeofday function allow measurements of real time, unlike other options such as the clock function which measures CPU time [88]. An example of the use of the timeval data type can be found in Listing 6.1.

To make sure that the correct times are measured, the start- and stop times will be measured right before, and after, the code blocks necessary to perform the database operation. In TC1 this would be just before starting the transaction, and after committing the transaction.

Listing 6.1: Using the timeval data type to measure elapsed time.

```c
int main () {
    struct timeval tval_before, tval_after;
    gettimeofday (&tval_before, NULL);
    sleep(1);
    gettimeofday (&tval_after, NULL);
    printf("Time in microseconds: %ld microseconds\n",
            ((tval_after.tv_sec - tval_before.tv_sec)*1000000L
            +tval_after.tv_usec) - tval_before.tv_usec);
    // Time in microseconds: 1003191 microseconds
    return 0;
}
```

One limitation when measuring execution times on the camera is how the Linux kernel schedules CPU time for the test case. As the test cases will run on a live system, where other processes will execute in parallel, there is a risk that measurements could be inaccurate due to CPU lock. To counter this the execution times will be measured for a large amount of iterations of each test case. The exact number of iterations will vary depending on the execution times.

6.5 Resource consumption

To measure resource consumption during the execution of the test cases the Linux iostat utility [89] and /proc files will be used. The iostat utility provides input/output statistics for the system, and will be used to measure read/write speeds towards the SD card, as well as CPU utilization. The /proc/$pid/status file will be used to gather memory statistics.

In Axis’ benchmark suite a tool was provided that measured the resource consumption using the above utility and file. To simplify the benchmarking process this tool will be used. This tool only has a resolution of one measurement each second, so to gather enough
measurements each test will be iterated multiple times. It was decided that the number of iterations will vary, so that the test case on the slowest database will take more than five seconds to complete.
6. Evaluation
Chapter 7
Results

In this chapter we will present the benchmark results for both Axis’ SQLite storage solution and the EJDB based solution presented in Chapter 5.

7.1 Execution times

Execution times are presented in tables with max, min, average (AVG) and standard deviation (STD) columns. These indicate the maximum, minimum and average execution times for the test case, as well as how much the execution times deviate from the average.

For those test cases that exhibited significant standard deviation relative to the average, the test cases were executed again after moving the database from the camera’s SD card to its internal flash storage. These additional tests were also performed for TC3 and TC4 to provide further insight into the variation of the averages.

![Figure 7.1: Execution time ratio for EJDB, compared to the normalized SQLite.](image-url)
7. Results

Figure 7.1 depicts the ratio between EJDB’s and SQLite’s execution times for all 17 test cases. The execution times have been normalized with respect to SQLite’s execution time (blue line). This translates to EJDB performing better (faster execution times) when times are below 100%, and worse (slower execution times) when times are above 100%. From the figure it can be observed that SQLite is only faster than EJDB in seven out of 17 test cases.

![Figure 7.1: Ratio between EJDB's and SQLite's execution times for all 17 test cases.](image)

An overview of how the standard deviation for the execution times varies for both databases is given in Figure 7.2. It can be seen that the standard deviation is quite large for the first two and last three test cases. The difference between these test cases and the others is that these include write operations.

![Figure 7.2: Standard deviation of test case execution times for SQLite and EJDB.](image)

7.1.1 Inserting new recordings

From the list of test cases in Section 6.3.2, only TC1 involves inserting new recordings into the database:

**TC1** Insert 1000 new recordings (entries) into a pre-populated database containing 10000 recordings.

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.0157</td>
<td>0.0014</td>
<td>0.0020</td>
<td>0.0009</td>
</tr>
<tr>
<td>SQLite</td>
<td>2.9303</td>
<td>0.9471</td>
<td>1.8451</td>
<td>0.6026</td>
</tr>
</tbody>
</table>

In Table 7.1 it can be seen that SQLite had significantly larger execution times when inserting new recordings into the database. Both EJDB and SQLite seem to suffer from a large standard deviation at 45% and 33% of the average, respectively. This led to TC1 being executed again with the database moved from the SD card to the camera’s internal flash (see Table 7.2). Moving the database resulted in a smaller standard deviation for
7.1 Execution times

Table 7.2: Execution times (s) when inserting one recording with the database on the camera’s internal flash (based on 1000 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.0085</td>
<td>0.0014</td>
<td>0.0018</td>
<td>0.0005</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.0278</td>
<td>0.0106</td>
<td>0.0131</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

both EJDB and SQLite at 3% and 8% of the average, respectively. It also significantly reduced the execution times for SQLite, while EJDB only saw minor improvements.

7.1.2 Updating recordings

From the list of test cases in Section 6.3.2, only TC2 involves updating recordings in the database:

TC2 Update the stop time for 1000 recordings (entries) in a pre-populated database containing 11000 recordings.

Table 7.3: Execution times (s) when updating one recording with the database on the SD card (based on 1000 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>3.5814</td>
<td>0.3617</td>
<td>0.4723</td>
<td>0.3655</td>
</tr>
<tr>
<td>SQLite</td>
<td>2.9314</td>
<td>0.0513</td>
<td>0.3776</td>
<td>0.4944</td>
</tr>
</tbody>
</table>

Table 7.4: Execution times (s) when updating one recording with the database on the camera’s internal flash (based on 1000 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.4079</td>
<td>0.3628</td>
<td>0.3877</td>
<td>0.0076</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.0173</td>
<td>0.0033</td>
<td>0.0043</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

Table 7.3 shows that there is no large difference between execution times as observed in TC1. Both databases still exhibited very large standard deviation at 77% of the average for EJDB, with the same value for SQLite being 130%. As previously, TC2 was executed again with the database moved to the camera’s internal flash (see Table 7.4). This resulted in a significantly smaller standard deviation for both databases at 2% and 26%, respectively, as well as shorter execution times for both databases. Just like in TC1, SQLite exhibited significantly shorter execution times, while EJDB only saw minor improvements.

7.1.3 Retrieving recordings

From the list of test cases in Section 6.3.2, TC3 up to TC14 all involve retrieving recordings from the database:

TC3 and TC4 Retrieve and sort all recordings (entries), based on start time, from a pre-populated database with 10000 recordings.
Table 7.5: Execution times (s) when retrieving all recordings with the database on the SD card, and sorting them based on start time (based on 500 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Sorted in ascending order</th>
<th>Sorted in descending order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>EJDB</td>
<td>1.6309</td>
<td>1.4295</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.7574</td>
<td>0.6574</td>
</tr>
</tbody>
</table>

Table 7.6: Execution times (s) when selecting all recordings with the database on the camera's internal flash, and sorting them on start time (based on 500 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Sorted in ascending order</th>
<th>Sorted in descending order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>EJDB</td>
<td>1.6726</td>
<td>1.4368</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.7787</td>
<td>0.6517</td>
</tr>
</tbody>
</table>

It can be seen in Table 7.5 that EJDB had roughly 145% longer execution times compared to SQLite, and that both databases exhibited very small standard deviation when sorting recordings. In Table 7.6 it can also be seen that moving the database from the SD card did not appear to increase performance when retrieving recordings from the database, as both the standard deviation and average are very similar.

TC5, TC6 and TC7 Retrieve all recordings from a pre-populated database with 10000 recordings, limiting the number of results.

Table 7.7: Execution times (s) when selecting all recordings with the database on the SD card, limiting the number of results (based on 4000, 2000 and 1000 measurements, respectively).

<table>
<thead>
<tr>
<th>Database</th>
<th>Limit</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>20</td>
<td>0.0058</td>
<td>0.0010</td>
<td>0.0014</td>
<td>0.0002</td>
</tr>
<tr>
<td>EJDB</td>
<td>100</td>
<td>0.0087</td>
<td>0.0024</td>
<td>0.0030</td>
<td>0.0004</td>
</tr>
<tr>
<td>EJDB</td>
<td>1000</td>
<td>0.0346</td>
<td>0.0206</td>
<td>0.0230</td>
<td>0.0019</td>
</tr>
<tr>
<td>SQLite</td>
<td>20</td>
<td>0.0144</td>
<td>0.0032</td>
<td>0.0038</td>
<td>0.0008</td>
</tr>
<tr>
<td>SQLite</td>
<td>100</td>
<td>0.0182</td>
<td>0.0070</td>
<td>0.0082</td>
<td>0.0013</td>
</tr>
<tr>
<td>SQLite</td>
<td>1000</td>
<td>0.0758</td>
<td>0.0505</td>
<td>0.0557</td>
<td>0.0025</td>
</tr>
</tbody>
</table>

In Table 7.7 it can be seen that SQLite had longer execution times compared to EJDB when retrieving recordings. At 20 and 100 recordings, SQLite had 170% longer execution times, but at 1000 recordings the percentage shrank to 140%.
For both databases it can be seen that the average increases together with the number of retrieved recordings. From 20 to 1000, the average execution time increased by a factor of 16 for EJDB, and 14 for SQLite.

As with TC3 and TC4 it is also observed that the standard deviation is very small.

**TC8, TC9 and TC10** Retrieve all recordings, based on an offset, from a pre-populated database with 10000 recordings, limiting the number of results.

<table>
<thead>
<tr>
<th>Database</th>
<th>Limit</th>
<th>Offset</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>20</td>
<td>100</td>
<td>0.0097</td>
<td>0.0015</td>
<td>0.0018</td>
<td>0.0005</td>
</tr>
<tr>
<td>EJDB</td>
<td>100</td>
<td>1000</td>
<td>0.0242</td>
<td>0.0137</td>
<td>0.0153</td>
<td>0.0015</td>
</tr>
<tr>
<td>EJDB</td>
<td>1000</td>
<td>5000</td>
<td>0.1863</td>
<td>0.0833</td>
<td>0.0885</td>
<td>0.0052</td>
</tr>
<tr>
<td>SQLite</td>
<td>20</td>
<td>100</td>
<td>0.0219</td>
<td>0.0048</td>
<td>0.0055</td>
<td>0.0009</td>
</tr>
<tr>
<td>SQLite</td>
<td>100</td>
<td>1000</td>
<td>0.0448</td>
<td>0.0231</td>
<td>0.0251</td>
<td>0.0019</td>
</tr>
<tr>
<td>SQLite</td>
<td>1000</td>
<td>5000</td>
<td>0.1947</td>
<td>0.1375</td>
<td>0.1437</td>
<td>0.0031</td>
</tr>
</tbody>
</table>

Table 7.8 shows that SQLite had longer execution times compared to EJDB when retrieving recordings. At 100 and 100 recordings, SQLite had 60% longer execution times, but at 20 recordings the percentage grew to 300%.

For both databases it can be seen that the average increases together with the number of retrieved recordings. From 20 to 1000, the average execution time increased by a factor of 49 for EJDB, and 26 for SQLite.

As with TC3, TC4, TC5, TC6 and TC7, it is also observed that the standard deviation is very small.

**TC11** Retrieve all recordings triggered by the event “ManualTrigger1” (2500 recordings) from from a pre-populated database with 10000 recordings.

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.2020</td>
<td>0.1064</td>
<td>0.1122</td>
<td>0.0053</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.2543</td>
<td>0.1710</td>
<td>0.1778</td>
<td>0.0036</td>
</tr>
</tbody>
</table>

It can be seen in Table 7.9 that SQLite had roughly 58% longer execution times compared to EJDB when retrieving recordings. As with TC8, TC9 and TC10, it is also observed that the standard deviation is very small.

**TC12, TC13 and TC14** Retrieve all recordings, using the start and stop times of the first and last recording, from a pre-populated database with 10000 recordings.

In Table 7.10, 7.11 and 7.12 it can be seen that EJDB had significantly larger execution times, compared to SQLite, and that both databases exhibited very small standard deviation. This is very similar to the results found in Table 7.5 where recordings were sorted on start time.
Table 7.10: Execution times (s) when selecting all recordings one hour later than the first recording’s start time with the database on the SD card (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>1.9265</td>
<td>1.7576</td>
<td>1.7801</td>
<td>0.0227</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.6504</td>
<td>0.5711</td>
<td>0.5864</td>
<td>0.0080</td>
</tr>
</tbody>
</table>

Table 7.11: Execution times (s) when listing all recordings one hour earlier than the last recording’s stop time with the database on the SD card (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>2.4806</td>
<td>2.2721</td>
<td>2.3032</td>
<td>0.0314</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.9368</td>
<td>0.8365</td>
<td>0.8591</td>
<td>0.0110</td>
</tr>
</tbody>
</table>

Table 7.12: Execution times (s) when listing all recordings one hour later than the start time and one hour earlier than the stop time of the first and last recordings respectively, with the database on the SD card (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>3.4331</td>
<td>3.1879</td>
<td>3.2182</td>
<td>0.0406</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.9067</td>
<td>0.7903</td>
<td>0.8174</td>
<td>0.0120</td>
</tr>
</tbody>
</table>

7.1.4 Removing recordings

From the list of test cases in Section 6.3.2, TC15, TC16 and TC17 involve the removal of recordings from the database:

TC15 Remove 100 recordings (entries), using their filenames (spread evenly across all recordings), from a pre-populated database with 10000 recordings.

Table 7.13: Execution times (s) when removing one recording with the database on the SD card, checking for unreferenced records (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>3.7213</td>
<td>0.3835</td>
<td>0.5533</td>
<td>0.5474</td>
</tr>
<tr>
<td>SQLite</td>
<td>3.3199</td>
<td>0.8088</td>
<td>1.4014</td>
<td>0.6732</td>
</tr>
</tbody>
</table>

Table 7.14: Execution times (s) when removing one recording with the database on the camera’s flash internal flash, and checking for unreferenced records (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.4544</td>
<td>0.3814</td>
<td>0.4189</td>
<td>0.0162</td>
</tr>
<tr>
<td>SQLite</td>
<td>0.8718</td>
<td>0.7431</td>
<td>0.7817</td>
<td>0.0190</td>
</tr>
</tbody>
</table>
EJDB performs better than SQLite with a 40% lower average when recordings were removed using the filename, see Table 7.13. However, this is overshadowed by the significant standard deviation at 98% and 48% of the average, respectively. As previously when encountering a large standard deviation, TC15 was executed again after moving the database (see Table 7.14). Moving the database resulted in a significantly smaller standard deviation for both EJDB and SQLite.

Additionally, one possible explanation for the larger average for SQLite was that the process of checking the tables for unreferenced records (described in Section 3.2) caused significant overhead.

This resulted in TC15 being executed again, without checking the tables for unreferenced records (see Table 7.15). By removing the additional table operations the new average was only one third of the previous.

**Table 7.15:** Execution times (s) when removing one recording with the database on the SD card, without checking for unreferenced records (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>1.8661</td>
<td>0.0565</td>
<td>0.4667</td>
<td>0.4739</td>
</tr>
</tbody>
</table>

**TC16** Remove all recordings from a pre-populated database with 10000 recordings.

**Table 7.16:** Execution times (s) when deleting all recordings with the database on the SD card (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>3.2713</td>
<td>0.1272</td>
<td>1.1016</td>
<td>0.9567</td>
</tr>
<tr>
<td>SQLite</td>
<td>11.5607</td>
<td>7.9449</td>
<td>9.6281</td>
<td>0.7137</td>
</tr>
</tbody>
</table>

**Table 7.17:** Execution times (s) when deleting all recordings with the database on the camera’s internal flash (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>0.0636</td>
<td>0.0313</td>
<td>0.0351</td>
<td>0.0037</td>
</tr>
<tr>
<td>SQLite</td>
<td>2.2393</td>
<td>2.0711</td>
<td>2.1269</td>
<td>0.0419</td>
</tr>
</tbody>
</table>

The execution times in Table 7.16 show a similar pattern to those found in Table 7.13 when removing a single recording, specifically the max, min and average execution times were very small compared to SQLite. The large standard deviation exhibited by EJDB (87% of the average) also resulted in TC16 being executed again after moving the database (see Table 7.17). Moving the database significantly decreased the standard deviation.

**TC17** Remove all recordings using the start time of the first recording, appended with 10 minutes, from a pre-populated database with 10000 recordings.

Unlike in TC15 and TC16, Table 7.18 shows that EJDB performed worse than SQLite when removing recordings, while filtering on start time. EJDB exhibited a 1.5 times larger max, 7 times larger min, and 1.45 times larger average.
Table 7.18: Execution times (s) when deleting recordings using the start time of the first recording, appended with 10 minutes, with the database on the SD card (based on 100 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>5.2933</td>
<td>1.1909</td>
<td>1.7897</td>
<td>0.7091</td>
</tr>
<tr>
<td>SQLite</td>
<td>3.6146</td>
<td>0.1587</td>
<td>1.2399</td>
<td>0.9993</td>
</tr>
</tbody>
</table>

7.2 Resource consumption

Resource consumption is presented using two plots: one that displays the CPU and memory usage over a time period, and another that displays the read and write speeds towards the SD card for the same time period. In the plots that show CPU and memory, the CPU usage is displayed as a continuous line while the memory usage is displayed only in short intervals. This is the result of measuring the memory usage of the test case only when it is running (no more than a few seconds), while CPU is measured continuously. In both the CPU and the read/write plots an initial spike can be observed. This spike is caused by the measuring tool when it starts, and should thus be ignored in any analysis.

Since the tool used for measuring resources only had a resolution of one measurement per second, TC1 and TC16 were hard to plot, as simply getting one measurement from EJDB would require more than 100 iterations, see Table 7.1. The same number of iterations would result in close to 3 minutes of execution time for SQLite, see Table 7.16.

Only selected plots that illustrate interesting results are found in this section. The remaining plots can be found in Appendix C.

7.2.1 Inserting recordings

From the list of test cases in Section 6.3.2, only TC1 involves inserting new recordings into the database:

**TC1** Insert five new recordings (entries) into a pre-populated database containing 10000 recordings.

![Figure 7.3: TC1 CPU and memory usage when inserting five recordings. CPU is represented by the blue line, while memory is represented by the green line.](a) EJDB (b) SQLite
In Figure 7.3 it can be seen that no information about memory usage can be found for EJDB. This is the result of the short execution times observed in Table 7.1. Two spikes in EJDB’s CPU usage, both close to 30%, can be observed at 15:55:16 and 15:55:22. These correspond to the first two spikes in write speed found in Figure 7.4. In Figure 7.4 it appears as though EJDB is writing information to its database in chunks, unlike SQLite which is continuously writing data. EJDB is also writing less information than SQLite.

### 7.2.2 Updating recordings

From the list of test cases in Section 6.3.2 only TC2 involves updating recordings in the database:

**TC2** Update the stop time for 20 recordings (entries) in a pre-populated database containing 10020 recordings.

![Figure 7.4: TC1 read/write speeds when inserting five recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.](a) EJDB (b) SQLite

![Figure 7.5: TC2 CPU and memory usage when updating 20 recordings. CPU is represented by the blue line, while memory is represented by the green line.](a) EJDB (b) SQLite
Figure 7.6: TC2 read/write speeds when updating 20 recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

In Figure 7.5 it can be seen that SQLite hovers around 55-60% CPU usage, while EJDB appears to hover just above 60%. It can also be seen that at 7300 KB, EJDB consumes roughly seven times the 1000 KB memory that SQLite uses. As previously EJDB also appears to write significantly less data to disk, and does so in chunks. However, this time SQLite also appears to write data in chunks.

7.2.3 Retrieving recordings

For all of the test cases that list recordings (TC5 to TC14), very similar measurements for CPU usage were observed for both databases (see Appendix C). EJDB and SQLite hovered around 60% CPU usage for most test cases, with the exception of TC7 in which EJDB had 70% CPU usage, so it can be assumed that the difference between the two databases is very small and hardly noticeable in most cases.

In all test cases EJDB consumed more memory than SQLite. The smallest difference observed was 2300 KB, compared to 1100 KB for SQLite, in TC9. The largest was 12000 KB, compared to 2400 KB for SQLite, in TC14. Additionally, in several EJDB test cases it seems that the memory appears to vary significantly throughout the execution of the test case. An example of such a test case can be found in Figure 7.7 where two drops in memory consumption were observed.

Additionally, in all the test cases where recordings were retrieved, it was observed that EJDB performed write operations (see Figure 7.8). This is an interesting observation, as SQLite performed no write operations when retrieving recordings.

7.2.4 Removing recordings

From the list of test cases in Section 6.3.2, TC15, TC16 and TC17 involve the removal of recordings from the database:

TC15 Remove 50 recordings (entries), using their filenames, from a pre-populated database with 10000 recordings.

In Figure 7.9 it can be seen that EJDB had roughly 10% higher CPU usage when compared to SQLite (70% for EJDB and 60% for SQLite). As with previous test cases
7.2 Resource consumption

Figure 7.7: TC4 CPU and memory usage for 10 iterations. CPU is represented by the blue line, while memory is represented by the green line.

Figure 7.8: TC4 read/write speeds for 10 iterations on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure 7.9: TC15 CPU and memory usage when removing 50 recordings. CPU is represented by the blue line, while memory is represented by the green line.
Figure 7.10: TC5 read/write speeds when removing 50 recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

EJDB consumed more memory than SQLite, approximately 44% more (computed using maximum values in Figure 7.9). In Figure 7.10 it can be seen that SQLite writes data continuously during the execution, while EJDB on the other hand writes data in chunks. Closer inspection of Figure 7.10 also suggests that EJDB writes more data to disk compared to SQLite as writes were performed in chunks and reach above 400 KB/s, but a rough estimate suggests that EJDB writes about 800 KB while SQLite writes more than 1000 KB.

TC16 Remove all recordings from a pre-populated database with 10000 recordings.

Figure 7.11: TC16 CPU and memory usage when removing all recordings. CPU is represented by the blue line, while memory is represented by the green line.

For TC16 no memory information could be observed (see Figure 7.11), due to the short execution times observed in Table 7.16. As in TC1, where similar behavior was observed due to the short execution times, the CPU top at 30% corresponds to the spike in write speed at 08:49:07.

TC17 Remove all recordings using the start time of the first recording, appended with 10 minutes, from a pre-populated database with 10000 recordings.
7.2 Resource consumption

![Figure 7.12: TC16 read/write speeds when removing all recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.](a) EJDB (b) SQLite]

![Figure 7.13: TC17 CPU and memory usage for 10 iterations. CPU is represented by the blue line, while memory is represented by the green line.](a) EJDB (b) SQLite]

In Figure 7.13 it can be seen that CPU usage for SQLite is roughly 60% while it varies between 60-80% for EJDB. EJDB also consumes significantly more memory than SQLite. The SQLite memory consumption appears to grow as the test executes, but the memory consumption at the end of the execution is still 6 times lower than EJDB, see Figure 7.13. As in the previous test case EJDB does not write data continuously to the SD card, but instead in two separate chunks (see Figure 7.14). A rough estimate suggests that both databases were writing 2500kB each.
Figure 7.14: TC17 read/write speeds for 10 iterations on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.
Chapter 8
Discussion

8.1 EJDB and SQLite execution times

EJDB exhibited significantly shorter execution times, compared to SQLite, when new recordings were inserted into the database. It also exhibited slightly shorter execution times in TC5 to TC10 where the result set was limited. However, the performance suffered in TC3 and TC4 where recordings were sorted on start time. In both cases the execution times reached above 1.4 seconds, which was 100% longer than SQLite. Test cases where recordings were filtered based on start- and stop times also exhibited similar results, and in TC14 the execution time for EJDB was 3.2s (200% longer than SQLite). It is very likely that a user would notice the difference in execution times, if the SQLite database was replaced with EJDB.

8.1.1 Conditional operations

By looking at Figure 7.1 it is observed that EJDB only exhibited longer execution times, compared to SQLite, in seven out of 17 test cases. In six, out of the seven, test cases conditional operations were encountered, namely in TC3, TC4, TC12, TC13, TC14 and TC17. In TC2 the longer execution times were only encountered before adding an index on the filename of the recordings (see Table 8.1).

In TC3 and TC4 recordings were sorted on start times, something which is accomplished by comparing start times using either the greater than (>) or less than (<) operators. The same applies to TC12, TC13 and TC14 where the operators are used directly in the queries.

To find out if conditional operations was really the bottleneck in these test cases, and not the SD card, TC3 and TC4 were executed again after moving the databases to the camera’s internal flash. Execution times were found to be almost identical (see Table 7.6), suggesting that it is in fact the conditional operators that results in long execution times.
8. Discussion

8.1.2 Impact of the SD card

In Figure 7.2 it was observed that TC1, TC2, TC15, TC16 and TC17 all suffered from very large standard deviation. All of these test cases performed write operations towards the SD card. In TC1, TC2, TC15 and TC16, moving the database from the SD card to the camera’s internal flash resulted in significantly smaller standard deviation. At the same time, moving the database to the internal flash and executing TC3 and TC4, where no information is written, showed that the standard deviation did not change. Therefore it could be concluded that the SD card is causing the large standard deviation.

It is also possible that the SD card is involved in the larger execution times for SQLite in TC1, TC2, TC15 and TC16 as well. In TC1 three different collections are modified in EJDB, but in two of them, i.e. recordings and blocks only new entries are inserted, this is probably written sequentially to the SD card. SQLite, however, stores the database as a single file, but this file consists of one or more pages [90]. It could be that tables in Axis’ current storage solution are stored in different pages inside the database file. Therefore it has to do random writes, which are much slower than sequential writes [91].

8.1.3 Remove recordings

In the two test cases that removed recordings, and did not involve start or stop times, EJDB performed significantly faster than SQLite. However, in TC17 the execution times were longer than SQLite.

In TC17 the diminished performance for EJDB is likely caused by the conditional operations, used when checking recordings start and stop time. Earlier it was mentioned that EJDB is not very efficient when performing such operations, and this would explain these results.

A likely cause for the longer execution times in TC15 when using SQLite is that the test case simulates the whole process described in Section 3.2, where additional SQL operations are performed to clear the database from unreferenced records. To validate this hypothesis, the test case was executed again (see Table 7.15) without performing the additional DELETE operations. In this test only recordings from the recordings table were deleted together with the corresponding blocks. The result showed that the average execution time dropped significantly, confirming the hypothesis that the large execution times were caused by the additional cleanup.

In TC16 the long execution times for SQLite cannot be credited to additional queries, as the SQL operation for emptying the database is simply eight DELETE * FROM SQL statements. It is unlikely that the number of tables is what makes SQLite take longer time, as EJDB is nine times faster. This suggests that EJDB is faster than SQLite at removing information from the database. As noted with the standard deviation, where TC16 was also covered, it is possible that this is caused by random writes when accessing pages.

Note that in SQLite the DELETE operation, without any WHERE statements, is optimized so that its performance is similar to that of a SQL TRUNCATE statement, which SQLite does not directly support [92].

8.2 EJDB and SQLite resource consumption

In all of the test cases it could be observed that EJDB had higher memory consumption compared to SQLite. In the worst case EJDB consumed 12% of the total memory (16 out
of 128MB) while SQLite only consumed 3%.

This is most likely related to the allocation of multiple variables in the process of building EJDB queries (see Section 5.3.3). Another possibility is that in EJDB, whole documents are retrieved from the database. The latter would suggest that retrieving a larger number of documents from the database would increase the memory usage. This is exactly what can be seen in TC5 and TC7 (see Appendix C), where the only difference is the size of the result set.

One problem with the read and write speeds of the SD card was that no data could be collected of the read speeds in any test case. A fast experiment where a recording, including footage, stored on the SD card was played in the camera’s Graphical User Interface (GUI) showed that read speeds could indeed be measured. Our only explanation for the lack of data is that the test cases requested insignificant amounts of data, and that it was overshadowed by the write speeds in the plots.

Another observation for EJDB was write operations, performed when documents were retrieved from the database. An example is depicted in TC4, Figure 7.8. In this test case recordings were simply retrieved and sorted. Still, it could be seen that EJDB wrote about 95kB of data in four separate chunks.

We do not have an explanation for these write operations, outside of EJDB writing data when it retrieves recordings. As they occur in every test case that retrieves information (see Appendix C) it is reasonable to assume that it is normal behavior for EJDB.

### 8.2.1 EJDB transactions and durability

One of the features of EJDB is durable transactions, but the pattern observed in write speeds (EJDB writes in chunks) caused these to only be somewhat durable. In the common sense transactions should be written to the database after they have been committed, but this was not the case in EJDB. In EJDB transactions were only written as part of the chunks observed in the write speeds. Because of this, it is possible to loose all information from several transactions by incurring power loss before the first chunk has been written.

Additionally, when the database was subjected to power loss during a transaction the database did not become corrupt. However, it became impossible to add additional records to the database, putting it in a passive state where information could only be read. The only way to restore the database to a functional state was to extract the information from the database and place it in a new copy.

### 8.3 Increasing performance in EJDB with indices

In both Table 7.3 and 7.4 EJDB performed worse compared to SQLite when updating recordings. We realized that in Chapter 5 we had introduced an index for the event id to speed up the retrieval of recordings. TC2 provides a very similar scenario where equality is checked for each recording id. Hence, one idea was to introduce indices on recording filenames. Table 8.1 shows the execution times when executing TC2 on the SD card, but this time with an index on recording filenames. Adding an index on the recording id significantly decreased execution times at the cost of 644kB to store the new index files created by EJDB. Thus, adding additional indices when one knows that the query will result in equality operations could provide additional performance to EJDB.
8. Discussion

### Table 8.1: TC2 with an index on recording id (based on 1000 measurements).

<table>
<thead>
<tr>
<th>Database</th>
<th>Max</th>
<th>Min</th>
<th>AVG</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJDB</td>
<td>3.5814</td>
<td>0.3617</td>
<td>0.4723</td>
<td>0.3655</td>
</tr>
<tr>
<td>EJDB</td>
<td>0.019</td>
<td>0.0015</td>
<td>0.0020</td>
<td>0.00079</td>
</tr>
</tbody>
</table>

#### 8.4 Building on Axis’ storage solution

In the current SQLite solution, data is normalized and numeric values are being used as primary keys in all the tables (see Section 3.1). In the ER model numeric primary keys are unnecessary as each tuple in the tables already contain unique fields that are good candidates for primary keys. For example the “filename” in the recordings and blocks tables, the “name” in the sources, events and actions tables and the “type” in the audios, videos and types tables. By replacing the numeric primary keys with the value of these fields it is possible to avoid the additional joins described in Section 3.1 and 3.2 when retrieving the source, event and action names.

A test was performed to measure the execution times with and without joins. In this test two databases were used: the original database, described in the ER model (see Figure 3.1), and one where all entities, except the blocks, were merged into a single entity. In the first database joins had to be performed to access all the recordings information, and in the second only a single database operation was needed. The test showed that execution times could decrease by up to 100% when avoiding joins.

Another improvement to the model could be to only retrieve the information that is actually needed. When performing recording playback, information such as “source name” and “recording type” are completely useless. If only necessary information is accessed, there would be no reason to join all the tables in the database. It would also make better use of the available bandwidth.

#### 8.5 Selecting a candidate

During the analysis in Chapter 4 several databases from different database paradigms were identified. However, most of them turned out to be unsuitable for the camera. Only two paradigms, key-value and document, were left after filtering the databases using Axis requirements (see Section 4.1).

An interesting question is whether we would have selected any other paradigm(s) if there were no requirements. As mentioned in Chapter 4, NewSQL and column-oriented databases were dismissed since they were too similar to RDBMS. The lack of relations between different entities in the ER model also makes the use of graph databases unsuitable as the number of edges would be small. It could have been interesting to look closer at the column family database, as it provides access to certain parts of the entities stored in the database. This would have allowed for a more user-tailored database solution where whole documents (or the equivalent) do not have to be retrieved. In the end, the choice would still be to choose a database based on aggregates, based on the analysis of how data is accessed (see Section 3.2).

As mentioned, graph databases were not very applicable due to the structure of the data. However, in the future the inclusion of metadata which could associate different recordings with each other, using objects and temporal information, could make this
paradigm a more interesting option. Today there are no graph databases aimed towards the embedded market, but with big data and metadata catching the interest of the industry, embedded-friendly graph databases might surface in the future.
Chapter 9

Conclusion

Today the choices of non-relational databases for use in embedded systems are very limited, as most of the available NoSQL databases focus on large and distributed applications. For the embedded camera system used in this thesis, only a few candidate NoSQL databases could be identified. The document database EJDB was found to be the most suitable database for evaluation against the SQLite database used in the camera.

In EJDB, data is stored as documents that can be grouped together in collections. As it is possible to query and index individual fields inside the documents, the database can be used in a similar manner as SQLite. However, one limitation with indices in EJDB is that they only increase performance when used as a part of an equality (=) operation, unlike in SQLite. EJDB also has limited support for the ACID concept, supporting basic transactions with durability. In tests, the durability proved to be insufficient for the camera, as power loss incurred in the middle of a transaction caused the database to end up in an inconsistent state. SQLite, on the other hand, has full support for ACID transactions and can survive power loss most of the time without affecting the state of the database.

For both EJDB and SQLite, the database is stored as plain files without any user authorization. This makes both databases portable, and allows for easy backing up by simply copying the database files. Additionally, the lack of authorization means that neither database can identify who is currently accessing the database.

SQLite, unlike EJDB, is very popular and has been optimized for low resource consumption. This is reflected in the amount of memory consumed by both databases, as EJDB in most situations consume significantly more memory than SQLite. EJDB’s memory consumption was at most 16 MB in the worst case, which is 450% higher than SQLite’s. However, both databases exhibit fairly similar load on the CPU of the system they run on. Use of EJDB is thus limited to systems which can support the larger memory usage, such as the Axis camera which was used for benchmarking (128MB RAM). When accessing database contents, EJDB exhibited longer execution times than SQLite, as soon as conditional operators (>, <) were involved. For other cases, such as insertion and removal of records/documents, EJDB proved to be more efficient than SQLite.

Both EJDB and SQLite are good candidates for use in an embedded system, and
selecting a database depends on the importance of different factors, such as memory usage, durability and execution times. In applications where database operations are primarily focused around insertion and removal of documents, EJDB provides an edge over SQLite in execution times, but at the cost of increased memory consumption. EJDB, as a document database, also lacks the restrictions that a relational schema incurs. This makes the database flexible and expandable as the schema can be modified without requiring database migrations. However, as surveillance cameras should require low maintenance and good durability, the lack of completely durable transactions in EJDB makes it unsuitable to replace SQLite in its current state. If this problem was to be fixed, it is possible that EJDB could be used in applications where user interaction is low.

9.1 Future Work

The two main obstacles that hinder the replacement of SQLite with EJDB are the lack of index support for conditional operations, and the problems with durability while writing information.

As an initial continuation of the work, the implementation of indices in EJDB could be investigated, in order to see whether support for conditional operators could be implemented. This might make it possible to speed up the filtering of recordings (see Section 3.2). As both EJDB and SQLite store indices in a B+ tree it should be possible to make EJDB as fast as SQLite, since it is much faster when only the equality operator is used.

As with indices, the durability concerns are also likely software-related. A good step towards shaping EJDB to be a good replacement for SQLite is to review the underlying storage engine and see whether it is possible to make it write data continuously.

Furthermore, the implementation of string comparisons using conditional operations is something to look into, as Axis’ cameras, and likely other solutions, rely on this functionality. Implementing this could make EJDB more popular.

During this thesis, there was not enough time to create a model for metadata, and as future work it is possible to look into a document based metadata model. This would provide additional insight into whether the document model could be suitable in future implementations of Axis’ (and other) storage solutions.
Bibliography


Appendices
Appendix A

NoSQL timeline

Figure A.1 contains a timeline for the appearance of NoSQL databases, according to the information found during the research phase of this thesis. It is hard to pinpoint exactly when the different paradigms started to surface, and the information depicted in the figure should not be considered as an absolute timeline. The purpose is to illustrate the rise of NoSQL in the 21st century.

Figure A.1: Database timeline.
A. NoSQL timeline
Appendix B
Cross-compiling EJDB

Prerequisites

1. CMake version 2.8.12+
2. EJDB version 1.2+
3. gcc version 4.7+
4. zlib-dev

Toolchain

To cross-compile using CMake a toolchain has to be used. Toolchains are files which contain specific build settings, such as which compiler to use, and can be supplied to CMake using -DCMAKE_TOOLCHAIN_FILE. The toolchain in Listing B.1 was used to compile EJDB for the P3367-V camera.

Listing B.1: toolchain-mipsisa32r2el.cmake

```cmake
# the name of the target operating system
SET(CMAKE_SYSTEM_NAME Linux)

# which compilers to use for C and C++
SET(CMAKE_C_COMPILER mipsisa32r2el-axis-linux-gnu-gcc)
SET(CMAKE_CXX_COMPILER mipsisa32r2el-axis-linux-gnu-g++)

# here is the target environment located
SET(CMAKE_FIND_ROOT_PATH $ENV(AXIS_TOP_DIR)/target/mipsisa32r2el-axis-linux-gnu)

# adjust the default behaviour of the FIND_XXX() commands:
# search headers and libraries in the target environment, search
# programs in the host environment
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
```
Compiling EJDB

Assuming the prerequisites have been resolved, and the toolchain file above is used to cross-compile EJDB, we provide the GNU makefile in Listing B.2 which can be used to compile and install EJDB. Note, that when running the make utility the current directory must be the root of the downloaded EJDB source.

Listing B.2: EJDB cross-compile makefile

```make
all: dirsetup
cd build; \# Change DCMAKE_INSTALL_PREFIX to compile for another
target environment
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$(AXIS_TOP_DIR)/target/mipsisa32r2el-axis-linux-gnu-
DCMAKE_TOOLCHAIN_FILE=toolchain-mipsisa32r2el.cmake ..;/
make
dirsetup:
  rm -rf build
  mkdir build
install:
  cd build; \make install
```

Errors

If the following error is encountered it is likely that the location where CMake should look for libraries is wrong. Make sure that the CMAKE_FIND_ROOT_PATH is correct in the toolchain.

```make
CMake Warning at src/CMakeLists.txt:61 (message):
  Library 'librt' not FOUND

CMake Error at src/CMakeLists.txt:66 (message):
  Library 'libm' not FOUND
```
Appendix C

Resource consumption plots

Figure C.1: TC1 CPU and memory usage when inserting five recordings. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.2: TC1 read/write speeds when inserting five recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.3: TC2 CPU and memory usage when updating 20 recordings. CPU is represented by the blue line, while memory is represented by the green line.

Figure C.4: TC2 read/write speeds when updating 20 recordings on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.
Figure C.5: TC3 CPU and memory usage when listing all the recordings in descending order. CPU is represented by the blue line, while memory is represented by the green line.

Figure C.6: TC3 read/write speeds when listing all the recordings in descending order from the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.7: TC4 CPU and memory usage when listing all the recordings in ascending order. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.8: TC4 read/write speeds when listing all the recordings in ascending order from the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.9: TC5 CPU and memory usage when listing all the recordings, limiting number of results to 20. CPU is represented by the blue line, while memory is represented by the green line.
**Figure C.10:** TC5 read/write speeds when listing all the recordings, limiting number of results to 20 from the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

**Figure C.11:** TC6 CPU and memory usage when listing all the recordings, limiting number of results to 100. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.12: TC6 read/write speeds when listing all the recordings, limiting number of results to 100 from the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.13: TC7 CPU and memory usage when listing all the recordings, limiting number of results to 1000. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.14: TC7 read/write speeds when listing all the recordings, limiting number of results to 1000 from the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.15: TC8 CPU and memory usage when listing all the recordings, limiting number of results to 20 and start from result 100. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.16: TC8 read/write speeds when listing all the recordings, limiting number of results to 20 and start from result 100.
The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.17: TC9 CPU and memory usage when listing all the recordings, limiting number of results to 100 and start from result 1000. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.18: TC9 read/write speeds when listing all the recordings, limiting number of results to 100 and start from result 1000. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.19: TC10 CPU and memory usage when listing all the recordings, limiting number of results to 1000 and start from result 5000. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.20: TC10 read/write speeds when listing all the recordings, limiting number of results to 1000 and start from result 5000. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.21: TC11 CPU and memory usage when listing all the recordings based on event name. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.22: TC11 read/write speeds when listing all the recordings based on event name. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.23: TC12 CPU and memory usage when listing all the recordings one hour later than the first recording’s start time. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.24: TC12 read/write speeds when listing all the recordings one hour later than the first recording’s start time. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.25: TC13 CPU and memory usage when listing all the recordings one hour earlier than the last recording’s stop time. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.26: TC13 read/write speeds when listing all the recordings one hour earlier than the last recording’s stop time. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.27: TC14 CPU and memory usage when listing all the recordings one hour later than start time and one hour earlier than stop time of the first and last recordings, respectively. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.28: TC14 read/write speeds when listing all the recordings one hour later than start time and one hour earlier than stop time of the first and last recordings, respectively. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.29: TC15 CPU and memory usage when 50 recordings are removed. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.30: TC15 read/write speeds when 50 recordings are removed. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.31: TC16 CPU and memory usage when all recordings are removed. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.32: TC16 read/write speeds when all recordings are removed. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.

Figure C.33: TC17 CPU and memory usage when all recordings that started before the first recordings start time, appended with 10 minutes. CPU is represented by the blue line, while memory is represented by the green line.
Figure C.34: TC17 read/write speeds when all recordings that started before the first recordings start time, appended with 10 minutes. The database is stored on the SD card. Write speed is represented by the blue line, while read speed is represented by the green line.
Relationsdatabasen har länge varit det dominerande paradigmet inom inbyggda system. En uppsjö av nya databasparadigm ställer dock frågan om det i dagsläget hade varit möjligt att avvika från industristandarden. Finns det något att vinna i att byta paradigm?

Inledning
Hos inbyggda system i dagens mjukvaruindustri är den så kallade relationsdatabasen en industristandard. För den storskaliga datainsamling som pågår inom områden som videoövervakning, social media etc. så stöter dessa databaser på problem när de utsätts för stora belastningar. Att data dessutom är under ständig förändring gör det också svårt för dessa databaser, eftersom de har begränsad flexibilitet. Utifrån detta har ett flertal databaser, under det gemensamma namnet NoSQL, utvecklats för att tillgodose dessa behov. Många av dessa riktar sig mot tjänster såsom Facebook, Twitter etc. men det vore intressant att även se om dessa typer av databaser har användningsområden inom videoövervakning och inbyggda system. Det vore trevligt om man kunde byta ut en relationsdatabas mot en mer flexibel NoSQL databas utan att förlora prestanda. Ett större utbud av användbara databaser vid utveckling av inbyggda system skulle också vara positivt.

Resultat
Det vi kom fram till med vårt arbete var att det är möjligt att använda en NoSQL databas i en övervakningskamera, men att urvalet är väldigt begränsat. Det visade sig att ett byte från en relationsdatabas till en dokumentdatabas i vissa fall kunde leda till ökad prestanda, men att dessa databaser inte är särskilt lämpade för inbyggda system och de krav som ställs på dessa. Till exempel så visade det sig att dokumentdatabasen EJDB var betydligt snabbare än relationsdatabasen SQLite på att lägga in (0.002s vs 1.8s) och ta bort (1s vs 9s) inspelningar i en databas. Dock så hade EJDB större resursofoforbrukning än SQLite, och i kameran så växtrade EJDBs minnesfoforbrukning mellan 1-12% medan SQLite låg mellan 1-3%. EJDBs filstorlek var också dubbelt så stor (10MB) som SQLite. Dessutom riskerade EJDB att förlora inspelningar och bli korrupt vid strömbrott på grund av hur den sparade information. Så även om det går att använda NoSQL i en övervakningskamera så är de databaser som finns i dag inte särskilt lämpade på grund av hållbarhetskrav och begränsade resurser.

Utvärderingsmetod
Det här examensarbetet, som utfördes på Axis Communications i Lund, fokuserade på att utvärdera en relationsdatabas mot en NoSQL databas. För att kunna välja en NoSQL databas var det viktigt att ha en bra bild över vad för information det är man vill spara, samt hur denna används. Detta ledde till en utvärdering av datamodellen i en av Axis övervakningskameror. Förutom detta så etablerade ett antal krav som skulle hjälpa oss att utesluta databaser som inte gick att använda på kameran. En undersökning av NoSQL databaser som kunde användas visade att det inte fanns särskilt många databaser som lämpade sig för användning inom inbyggda system, och endast ett fåtal av dessa erbjudde samma funktionalitet som en relationsdatabas. Efter att ha valt ut en databas så designades ett antal testfall baserat på hur Axis använder sin datamodell. Därefter utfördes mätningar av exekveringstider och resursofoforbrukning (CPU och minne) för dessa testfall.