Faculty of Electrical Engineering,
Mathematics & Computer Science

Synthesis and Development of a Big Data architecture for the management of radar measurement data

Alex Aalbertsberg

Master of Science Thesis
November 2018

Supervisors:
dr. ir. Maurice van Keulen (University of Twente)
prof. dr. ir. Mehmet Akşit (University of Twente)
dr. Doina Bucur (University of Twente)
ir. Ronny Harmanny (Thales)

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

THALES

Approval Internship report/Thesis of:

Alexander P. Aalbertsberg
………………………………………………………………………………………………………

Title: …Synthesis and Development of a Big Data architecture for the management of radar measurement data

Educational institution: …University of Twente……………………………………………………..

Internship/Graduation period:…2017-2018……………………………………………………………..

Location/Department:…435 Advanced Development, Delft……………………………………

Thales Supervisor:……R. I. A. Harmanny………………………………………………………

This report (both the paper and electronic version) has been read and commented on by the supervisor of Thales Netherlands B.V. In doing so, the supervisor has reviewed the contents and considering their sensitivity, also information included therein such as floor plans, technical specifications, commercial confidential information and organizational charts that contain names. Based on this, the supervisor has decided the following:

- ○ This report is **publicly available (Open).** Any defence may take place publicly and the report may be included in public libraries and/or published in knowledge bases.

- ◉ This report and/or a summary thereof is **publicly available to a limited extent (Thales Group Internal)**. for 2 years. After 7-9-'20 it is "Open". It will be read and reviewed exclusively by teachers and if necessary by members of the examination board or review committee. The content will be kept confidential and not disseminated through publication or inclusion in public libraries and/or knowledge bases. Digital files are deleted from personal IT resources immediately following graduation, unless the student has obtained explicit permission to keep these files (in part or in full). Any defence of the thesis may take place **in public to a limited extent**. Only relatives to the first degree and teachers of the ……………n/a……………………department <*name department*> may be present at the defence.

- ○ This report and/or a summary thereof, is **not publicly available (Thales Group Confidential)**. It will be reviewed and assessed exclusively by the supervisors within the university/college, possibly by a second reviewer and if necessary by members of the examination board or review committee. The contents shall be kept confidential and not disseminated in any manner whatsoever. The report shall not be published or included in public libraries and/or published in knowledge bases. Digital files shall be deleted from personal IT resources immediately following graduation. Any defence of the thesis must take place **in a closed session** that is, only in the presence of the intern, supervisor(s) and assessors. Where appropriate, an adapted version of report must be prepared for the educational institution.

Approved:

(Thales Supervisor)

Approved:

(Educational institution)

Delft, 7 September 2018

(city/date)

THALES

(copy security)

This document is not to be reproduced, modified, adapted, published, translated in any material form in whole or in part nor disclosed to any third party without the prior written permission of Thales.

©Thales 2016 All Rights Reserved

Abstract

This research project proposes an architecture for the structured storage and retrieval of sensor data. While the demonstrator described has been developed in the context of Thales radar systems, different applications can be considered for certain classes of companies, specifically the ones that also deal with sensor data from many different machines and other sources. This demonstrator makes use of a distributed cluster architecture commonly associated with big data systems as well as software from the Apache Hadoop ecosystem.

The requirements from Thales dealt with a few different actions that needed to be able to be performed by the end users of the system. These actions involved the ability for the system to ingest data from log files and streaming data sources, as well as the ability for end users to query and retrieve data from the distributed storage. Research has been performed in order to decompose the requirements from Thales into a set of technical problems, which were then solved by making an inventory of technologies that can deal with these problems. By implementing the demonstrator, it became possible to store sensor data and retrieve it.

Preface

Dear reader,

Before you lies the culmination of the past year of work I have performed at Thales. It will serve as my Master's Thesis for the study Computer Science, Software Technology specialization at the University of Twente. Of course, such a task cannot be completed by the hands of a single person alone. As such, there are a few people I would like to personally thank.

Firstly, I would like to thank Maurice van Keulen, Mehmet Akşit and Doina Bucur, my team of supervisors at the University of Twente for their invaluable input over the course of the project.

Secondly, I would like to thank my supervisors at Thales: Ronny Harmanny for overseeing the project and steering me in the right direction, and Hans Schurer for overseeing my daily work and providing input where necessary, and of course my thanks goes out to Thales as a whole for providing me with a place to perform this fun and challenging project.

Finally, I would like to thank the people that stand closest of all to me. My deepest gratitude goes out to my mom Annelies and my dad Fred, who continued to support and believe in me.

I hope you will enjoy reading my thesis.

Alex Aalbertsberg

Emmen, November 16, 2018

Glossary

adaptability Ability to adjust oneself readily to different conditions.[1] In the context this project, it refers to the ability of the System Under Development to adapt to the storage and processing of differing data formats.

cluster computing A group of computers that are networked together in order to perform the same task. In many aspects, these computers may be seen as a single large system.

columnar database A database that stores data by column rather than by row, which results in faster hard disk I/O for certain queries.

Data Definition Specification Specification that describes the format that data of a certain type should adhere to.

distributed processing The execution of a process across multiple computers connected by a computer network.

machine learning An application of artificial intelligence (AI) that allows a program to perform actions, handle outside impulses and teach itself how to behave without explicitly programming it to do so.

Master node A master node is the controlling node in a big data architecture. The responsibility of such a node is to ”oversee the two key functional key pieces that make up a cluster”: cluster data storage and cluster computing.

pattern recognition The process by which a computer, the brain, etc., detects and identifies ordered structures in data or in visual images or other sensory stimuli [2].

plot A processed form of raw measurement data, a plot contains the location of a detected entity [3, pp. 118–119].

raw measurement data Measurement data before it has been processed. This type of data is very large in size.

relatability In data science, data relatability is the ability to create logical relations between different records of data [4, p. 17].

Shadow Master node A copy of the master node which is able to run all processes of the main master node in case of a master node failure. In Hadoop, it is also known as a NameNode fail-over.

slave node A slave node is a cluster node on which storage and computations are performed.

SQL Structured Query Language. Provides a DSL that allows for the querying of structured data.

track A track is a collection of plots, which together constitute the movement and location of an entity over time [3, pp. 118–119].

Contents

Abstract i

Preface iii

Glossary v

1 Introduction 1
1.1 Motivation . . . 1
1.2 Goals . . . 2
1.3 Approach . . . 2
1.4 Structure of the report . . . 3
1.5 Contributions . . . 4

2 High-level specification 5
2.1 Requirements . . . 5
2.1.1 Objective . . . 5
2.1.2 Main requirements . . . 6
2.2 Rationale . . . 6
2.3 Related work . . . 10

3 Synthesis of a sensor data management system 13
3.1 Synthesis overview . . . 15
3.2 System architecture . . . 16
3.2.1 Architecture decomposition . . . 16
3.2.2 Storage Mechanisms . . . 17
3.2.2.1 Functional Requirements . . . 18
3.2.2.2 Non-Functional Requirements . . . 19
3.2.2.3 Architecture diagram . . . 19
3.2.2.4 Open questions . . . 20
3.2.3 Extract, Transform and Load (ETL) . . . 21
3.2.3.1 Functional Requirements . . . 21
3.2.3.2 Non-functional Requirements . . . 22

3.2.3.3 Architecture diagram 23
3.2.4 Querying 24
3.2.4.1 Functional Requirements 24
3.2.4.2 Non-functional Requirements 25
3.2.4.3 Architecture diagram 26
3.2.5 Adaptability 26
3.2.5.1 Functional Requirements 26
3.2.5.2 Non-functional Requirements 27
3.2.5.3 Architecture diagram 28
3.2.6 Access Control 28
3.2.6.1 Functional Requirements 28
3.2.6.2 Non-functional Requirements 29
3.3 Project objectives 31

4 Existing technologies **33**
4.1 Big Data in General 33
4.1.1 Big data challenges 33
4.1.2 Hadoop KSPs 35
4.1.3 Apache Hadoop 35
4.1.3.1 Master-Slave architecture 35
4.1.3.2 Hadoop processing layer: MapReduce 38
4.2 Storage mechanisms 40
4.3 Adaptability 41
4.4 Cryptography 44
4.5 ETL and ELT 46
4.5.1 Apache Storm 48
4.5.2 Apache Kafka 49
4.5.3 Apache Flume 50
4.5.4 Apache Hive 51
4.5.5 Apache Spark 52
4.5.6 Apache Sqoop 53
4.6 Querying 53
4.6.1 Apache Spark 54
4.6.1.1 Spark DataFrame Example 55
4.6.1.2 Spark Dataset Example 56
4.6.2 Apache Hive 57
4.6.3 Apache Phoenix 57

5 System implementation **59**
5.1 Storage Mechanism 59

5.1.1 HBase 60
5.1.2 MySQL 63
5.2 ETL 64
5.2.1 Prototype 1: Apache Kafka + Apache Storm 64
5.2.2 Prototype 2: Apache Spark 69
5.3 Querying 70
5.3.1 Prototype 1: Spark RDD Prototype 70
5.3.2 Prototype 2: Spark DataFrame Prototype 72
5.3.3 Prototype 3: Spark DataSet Prototype 74
5.4 Adaptability 76
5.5 Access Control 77

6 Validation 79
6.1 Cluster setup 79
6.2 Requirements validation 80
6.2.1 Storage 80
6.2.2 ETL 82
6.2.3 Querying 83
6.2.4 Adaptability 84
6.2.5 Access Control 85
6.3 Experiments 85
6.3.1 ETL performance experiment 86
6.3.1.1 Apache Storm 86
6.3.1.2 Apache Spark 86
6.3.1.3 Results 86
6.3.2 Querying performance experiment 87

7 Conclusions 89
7.1 Discussion 90
7.2 Future Work 91

References 92

Appendix A Collection of Hadoop-supporting Apache projects 97

Appendix B Converter Function 103

List of Figures

3.1 The synthesis diagram of the radar information system. 14
3.2 High-level architecture of the system. 17
3.3 Storage architecture diagram 19
3.4 ETL architecture diagram 23
3.5 Querying architecture diagram 26
3.6 Adaptability architecture diagram 28

4.1 A master-slave architecture as it is used in Apache Hadoop. 36
4.2 An overview of the inner workings of a MapReduce job. 40
4.3 An overview of the structure within CP-ABE. 45
4.4 An overview of the process within CP-ABE. 46
4.5 An example of ETL with Apache Kafka and Apache Storm. 47
4.6 General example of a Storm topology. 49
4.7 An overview of the standard Flume data flow mechanism 50
4.8 An overview of the Flume aggregated data flow mechanism 51

5.1 HBase Table Design 61
5.2 MySQL Table Design 64
5.3 ETL Implementation Architecture Overview 65
5.4 Overview of prototype implementation of Kafka and Storm for ETL . . 65
5.5 Querying Implementation Architecture Overview 70
5.6 Spark RDD Implementation Overview 71

List of Tables

3.1 Storage Functional Requirements 18
3.2 Storage Non-Functional Requirements 19
3.3 ETL Functional Requirements 21
3.4 ETL Non-Functional Requirements 22
3.5 Querying Functional Requirements 24
3.6 Querying Non-Functional Requirements 25
3.7 Adaptability Functional Requirements 27
3.8 Adaptability Non-Functional Requirements 28
3.9 Access Control Functional Requirements 29
3.10 Access Control Non-Functional Requirements 29

4.1 Sample relational table [29] 41

6.1 Cluster Hardware Description Table 80
6.2 ETL performance experiment results 86
6.3 Query performance experiment results 87

Chapter 1

Introduction

With an increasingly large amount of data being generated at a growing pace by modern-day computer systems comes the need for technologies that are capable of handling this growth. Big data is still very much in its infancy, and corporations are only just beginning to realize the potential of these technologies.

This thesis has been written at Thales Netherlands, a corporation that mostly works on naval defense systems, a prime example being radar technology. Some other areas of business would be air defense, cryogenic cooling systems and navigation systems.

1.1 Motivation

Companies across all industrial fields deal with an increase in the amount of data generated by their business operations. This increase calls for the use of smart technologies in order to manage this data. In many cases, this amount of data cannot be persisted in a single physical system. In order to solve this, cluster computing is utilized to ensure that data storage can occur on a system consisting of multiple computers with the same goal. Examples of classes of companies that may deal with these problems could be hospitals or manufacturing plants.

The problem Thales faces at this point in time is as follows: radar systems generate a large amount of sensor data. Thales logs this data regularly and stores the data on a carrier (a CD/DVD, a USB drive, a hard drive). As a result, data from different logging events is scattered across many different carriers. Thales uses this logged data for various purposes, such as analyzing the performance of their own radar applications or developing algorithms. However, as a result of the scattering of data, it is currently difficult to retrieve and use this data efficiently.

This is something that Thales would like to change. The resulting assignment is to research and implement a data storage and processing system that is able to

store these large volumes of sensor log data. It should also be possible to query this data, so that researchers are able to retrieve (sub)set(s) of the data they would like to use. It should be noted that the radar applications themselves fall outside of the scope of this project.

1.2 Goals

The goal of the project is to design and implement a demonstrator that aims to have a uniform data storage, i.e. a form of storage that grants the opportunity to persist data from different data sources, to correlate between different types of data sources, and to extract data in a uniform format, to expedite the process of working with this data.

The research question can be formulated as follows: *What is a way to develop a big data system in which sensor data can be stored and uniformly queried, and which technologies are most suitable to perform the storage and querying of this data?* We will decompose this research question down into questions that directly impact each research area further in this thesis.

1.3 Approach

The approach to deal with this project is as follows: Firstly, a set of initial requirements has been set up for the project. This set of requirements has evolved several times over the course of the project. For the purpose of this project, we chose to take a big data approach. This is because of a few factors, such as potentially required storage size, required processing power and the formats of different stages of radar data. To clarify this last fact, sensor data from a radar is a very broad definition in this project, as it can vary from fully processed tracking data, which is relatively small, to raw radar measurement data, which is relatively large.

After setting the initial requirements, a literature research is conducted on the fields of big data and related topics. Some of these topics are generic researches, whereas some others are specific to the requirements set by Thales. From the results of the literature research, it becomes possible to apply the process of synthesis to create an architecture for the sensor data management system.

Synthesis entails diving deeper into the domain-specific requirements and solutions available for each individual part of the system. From this, a set of tools that can be used to design and implement the system that Thales desires can be determined.

From this, it is also possible to determine which parts of the system might require custom implementation, as well as which parts will work as desired out-of-the-box.

The next step in the project is to design the system, starting with a design for the data model. This entails designing how to structure data in the chosen storage mechanism, and choosing whether to store all of the data within the big data architecture, or whether it might be more beneficial to store administrative data elsewhere. Another part of the design is selecting tools that will be used to implement a demonstrator.

The demonstrator will consist of a combination of selected technologies that can perform required operations with regards to sensor data, the design of the data model and any required custom implementations for parts of the system that do not work out of the box. Finally, the demonstrator will be implemented.

At the end of the project, the demonstrator will need to be validated. This is done by comparing the functionalities of the demonstrator against the requirements that were set at the start of the project, as well as by executing a few comparative experiments that will help decide which technology suits best in case there are multiple available technologies that are capable of fulfilling the same role in the demonstrator.

1.4 Structure of the report

Firstly, the high-level requirements for this project will be determined in Chapter 2. In Chapter 3, the high-level requirements will be decomposed into required subsystems. In this chapter we will also use the results of literature research combined with the requirements to design a preliminary form of the system architecture. The available technologies for each individual part of the system will be described and compared in Chapter 4. In Chapter 5, we will look at the implementation details for each individual part of the system, and describe how they function. Chapter 6 will deal with the validation of the demonstrator, by performing requirements validation and several comparative experiments. Finally, Chapter 7 will list conclusions about the project and discuss its results, as well as list recommendations to Thales for future research.

1.5 Contributions

The contributions in this report are fourfold. The most important contributions that this project has made are the decomposition of requirements into a high-level architecture with accompanying reasoning. The requirements set by Thales are used to perform the process of synthesis, which results in an architectural design consisting of required system components.

This design consists of components whose requirements can be satisfied by making choices between different types of available software. One of the contributions in this report is the reasoning and choices made between different types of software and how they fit in the overall architecture as one of the system components.

Additionally, the report will describe the implementation details of each of these software choices. This will describe exactly how each of the functional requirements are fulfilled, such that end users can use the system as desired.

Finally, one of the contributions of the chosen architecture may be the societal benefits that it delivers. Since the overall design is relatively generic and describes how to deal with information from many different sources, it may prove useful to any other type of company that deals with a similar problem: high volumes and throughput of data from many different sources.

The focus of the project was mostly to deliver on the first three contributions, in order to translate Thales requirements into a fully functional demonstrator, which shows that the desired situation is possible and viable.

Chapter 2

High-level specification

As mentioned in the introduction, the current situation at Thales is that sensor log files are stored in separate data carriers, and there is no simple way in place for researchers working at Thales to efficiently use (combinations of) these datasets. Therefore, they would like to have a system researched that would allow them to do this. The main task of such a system is to allow for the storage and extraction of (parts of) sensor data, primarily being radar data. This section will describe the specification of such a system, by delineating the main objective of the system, and the high-level requirements that come forth from this objective.

2.1 Requirements

The following section will describe the requirements that have been set for this project. We will start with the ultimate objective of the system, and then break that down into individual requirements, as well as explain the rationale behind each requirement.

2.1.1 Objective

The main objective of the System Under Design (SUD) is to grant the ability to store and retrieve sensor log data, initially for the purpose of analysis by Thales researchers. These researchers will use the data to improve the capabilities of radar applications. Another application of the system would be to grant the ability to reconstruct a situation based on data stored in the database, and to use the reconstructed situation as evidence, for analysis or for training purposes. It needs to be possible to persists data on the system in an encrypted format. Encrypted data should only be accessible by people that have the correct access permissions.

2.1.2 Main requirements

The main requirements are requirements that need to be met in order for the system to be able to complete the objective. These requirements are as follows:

1. The system must store measurement information from radars and other sensors. In the context of this project, these sensors will be known as data sources.
2. The system must be able to ingest data from existing log files.
3. The system must be able to ingest data from streaming sources.
4. The system must be capable of containing permissions in order to limit user access to specific parts of the database.
5. The users must be able to retrieve information from the system via queries, both from a single data source, as well as cross data source.
6. The system must be capable of handling differences in terms of storing data with different formats. This will be referred to as adaptability.

2.2 Rationale

In the following sections, the rationale for each of the individual requirements will be explained.

Requirement 1: The system must store measurement information from radars and other sensors.

We are trying to store a large amount of sensor data for several purposes. In this section, we will attempt to delineate properties of the data we are trying to store, so that we can identify potential technical problems that need to be solved for the project to be successful. So far, the following properties of the data are known:

- We will deal with a growth of incoming measurement data. The solution will need to accommodate for different types of sensor data.
- The size of specific parts of the data set will depend on the types of data being stored. Using radar as an example, raw measurement data can consist of multiple gigabytes of data per second, whereas tracks will be much smaller, as it is processed data.

- There are multiple known formats for sensor data, thus we are dealing with structured data. These formats can vary in terms of the information stored in them, and each individual data field can store data in a different unit compared to other data types. For the purpose of querying these and retrieving uniform data, adaptability needs to be applied to the solution (See Requirement 6).

Other examples of sensor data stored in the system besides radar data could be the following:

- Meteorological information (Weather at a certain time, wind speed, temperature, etc.)
- The type of radar used, and its associated format
- Camera data
- Radar state
- Rotational state
  - Location
  - Pitch (Angle of the ship, may vary due to waves etc.)
- Commands, button presses and any other actions that are loggable

To accomplish the creation of an architecture that will support the properties listed above, there are some techniques and technologies available that facilitate them.

This data set will be used for the improvement and testing of a certain radar applications. Some examples are:

- A machine learning algorithm for pattern recognition. This will include a trainer that will be taught to classify entities by their coordinates and other information.
- Linking data into MATLAB. MATLAB is an environment that can accept (or be taught to accept) many different types of data.
- A tool that can reconstruct situations accurately based on recorded data. By combining the radar data with other sensor data, it is possible to create an overview of a situation that is as complete as possible.

Requirement 2: The system must be able to ingest data from existing log files.

As mentioned in the introduction, sensor data is currently logged into large and cumbersome log files. These files are then stored on a data carrier. Thales wants to have a solution that makes it possible to transfer log files from these old data carriers, and store it in a format where it is possible to query the data efficiently. Since we are transferring this particular type of data in bulk, we will need to use batch processing in order to store this data in our storage mechanism.

Requirement 3: The system must be able to ingest data from streaming sources.

In addition to being able to transfer old data from carriers to the system, Thales would like to have the possibility to perform live measurements to test their own radar systems. They would like to be able to record sensor data as mentioned in Requirement 1 in live situations.

Radar systems constantly emit messages about their state and their measurements. This constant stream of information will require the use of stream processing in order to prepare it for storage in the system. As mentioned in Requirement 1, these messages may consist of raw measurement data, i.e. all emitted signals and their reflections, but may also consist of processed plot or track information.

Requirement 4: The system must be capable of containing permissions in order to limit user access to specific parts of the database.

Requirement 4 It is undesirable for measurement data to fall into the wrong hands. Because of this, data in the system should adhere to a certain level of data security. This security will need to take place on multiple levels:

- Network level. The network that hosts the system will need to be secured, such that outsiders cannot break into it.
- Machine level. Machines themselves should be protected, such that only people with the correct access permissions can access it.
- Data level. (Parts of) the data should only be viewable by those who have the correct access, and queries should only return the data if correct access credentials have been provided.

Since network-level and machine-level security are environment-specific security properties for this particular assignment, Thales will take care of these properties, which leaves the focus on data security.

As mentioned, measurement data should be treated as confidential, and therefore it needs to be protected from any unwanted source attempting to access it. It should also not be possible to tamper with data, so there should be no way for anyone to change anything about the measurement data, i.e. it should be read-only.

Requirement 5: The users must be able to retrieve information from the system via queries, both from a single data source, as well as cross data source.

The system should allow users to query the data set. The extracted data may be used for previously mentioned applications, such as situation reconstruction and pattern recognition software. Queries should allow users to specify filters, as well as filter values and/or ranges to narrow down the search results as accurately as possible. Because of the vast amount of data that will be stored in the system, it seems highly likely that queries will need to be run across the data set in a distributed fashion.

There are a few different ways to approach this. In current mainstream software development, structured data is usually queried with the help of the Structured Query Language (SQL). The problem with this approach is that traditional database management approaches do not scale very well as the data set grows larger. There are big data techniques available for efficiently searching through data, should we choose to adopt that architecture. These querying techniques will require a data access point to be present at the server side, whether that be an SQL server, or a functional API.

Requirement 6: The system must be capable of handling differences in terms of storing data with different formats. This will be referred to as adaptability.

As mentioned in Requirement 1, the data that originates from sensors can differ in terms of data format. During interviews, we have established that it should be possible to query data from multiple data sources at once. There needs to be a way to correlate data between different data sources.

To clarify the correlation of data, consider this example: Let A, B be data sources for the system. Data source A consists of a field set height,weight, and data source

B consists of a field set height,weight. Consider both fields *height* to contain the same information, stored in the same unit. These two fields are considered to have an *Equality* relation towards each other. An *Equality* relation is a relation where two fields are equal both in terms of the information they give, as well as the unit they are stored in.

Now, consider the fields *weight* between data sources A and B to contain the same information, but stored in a different unit. For example, field

$$weight_A$$

might store someone's weight in kgs, whereas field

$$weight_B$$

might store it in lbs. These two fields are considered to have a *Convertible* relation to each other. A *Convertible* relation is a relation where two fields contain the same information stored in a different unit. An arithmetic conversion can take place to unify these fields under the same unit.

To facilitate this requirement, we will need a way to define these relations in our solution, as well as add intermediate steps to make sure that data is uniform when it is queried. We will be gathering data from many different sources. Each of these sources may correspond to a different data format, each of which will need these intermediate steps defined specifically for their format.

An example for radar sensor data is as follows: the data coming from a radar can be split up into three categories: Raw measurement data, processed data containing a single detection of an entity, which also called a plot. Finally, there is a data structure that contains a collection of plots that together show the movement of an entity over the course of time. This is referred to as a track. Each of these categories can contain different information and units of storage compared to categories of other sensor types.

2.3 Related work

This section will mention some other projects where people have attempted to design and/or implement something similar to the constraints set for this project. While research related to radar data in particular may not always be public, it is still possible to find research that deals with data that has a similar structure or similar constraints.

Sangat et al. propose a framework for the ingestion and processing of sensor data in manufacturing environments. Factories contain machinery that generate a lot of different kinds of sensor data, all of which can be used to further improve

the manufacturing process. They propose a solution that involves the combination of MongoDB and the MongoDB Connector for Spark, which proved to be a more efficient method to ingest data than MongoDB's native ingestion tool. [5]

Hajjaji and Farah have conducted research into the performance of certain NoSQL databases with regards to remote sensing technologies coming from satellites. As one can imagine, these systems generate huge amounts of image data per time unit, making a distributed storage solution preferable. In particular, they compare Apache Cassandra, Apache HBase and MongoDB. For their specific solution, Cassandra turned out to be the most suitable storage solution. [6]

In their research, Manogaran et al. describe an architecture for a big data ecosystem for dealing with Internet of Things and healthcare monitoring data. They use Apache Pig and Apache HBase for the respective collection and storage of data generated from different sensor devices. In addition to this, they also consider a model for different security classifications of data. This model consists of a key management service combined with a way to categorize data in terms of sensitivity. [7]

Luo describes a Hadoop architecture for storing data related to smart campuses. The work contains a precise description of cluster hardware, along with the used software (Apache HBase and Apache Hive). [8] Shen et al describe an enhancement to HBase's native capabilities, allowing for multi-conditional queries. HBase does not support this natively, while it is a common operation found in relational database queries. [9] Fan et al use traffic data and store it in Hadoop in order to support a machine learning process, in order to predict the time it will take to travel from one point to another. [10]

As can be construed from this section, there are many different fields that may benefit from the use of big data software, and many different fields can benefit from a structured way to delineate such an architecture based on specific requirements. There is no real ”standard” way to do big data: There are many different possibilities when it comes to storage, processing and middleware solutions, and cherry-picking the best ones seems to be the way to go.

Chapter 3

Synthesis of a sensor data management system

In order to solve the stated problem, the sensor data management system must be constructed. The requirements for the system have been discussed in Chapter 2. From these requirements we can derive technical problems that need to be addressed in order to successfully solve the main problem. The main problem statement is composed of the following technical problems:

1. **Storage efficiency problem**. Radar data is currently stored in large log files on common data carriers. Thales would like to see a system where this data can be stored in a single system. The data should be persisted in a sort of sensor data repository, where data can be queried and correlated among different sensor types.

2. **Incomplete situational data problem**. The data that comes from different sensors from the same measurement moments needs to be stored as well, so that a complete overview of the situation can be constructed from the retrieved data.

3. **Scalability problem**. With data sets growing ever larger and processing capabilities not growing quickly enough to linearly handle the data growth, a scalability problem is born. In order to be able to store and process large and ever-growing amounts of data, an entirely new software architecture will be required. Such a system is not yet in place at Thales at the moment, and thus we will need to design and implement this new architecture from scratch.

4. **Data access problem**. We also want to be able to access or query the data we store. Research needs to be done to determine which techniques provide functionality that allows the data to be queried, while also keeping the solution scalable.

5. **Varying data format problem**. Data from different types of radars often does not conform to the same data standard. Certain fields may contain related information in a different format or unit. The system must be capable of dealing with data containing similar information (in different formats), and be able to distinguish or translate freely between different formats.

6. **Data security problem**. There needs to be a distinction between the access rights of different end users of the system. Some users might only be allowed to read the contents of certain sets of sensor data, whereas others might be allowed to see all data sets, or maybe even write to certain data sets.

In order to find the solutions to these problems, we will use the process of synthesis. We will need to identify abstract solution domains that correspond with our problems. By exploring the techniques with these solution domains, we should be able to define a set of concrete solutions that will be able to satisfy the requirements of this project. [11] The rest of this chapter will define technical problems, underlying subproblems, relevant abstract solution domains and abstractions that were derived from the synthesis process. Figure 3.1 shows a diagram depicting the results of this process.

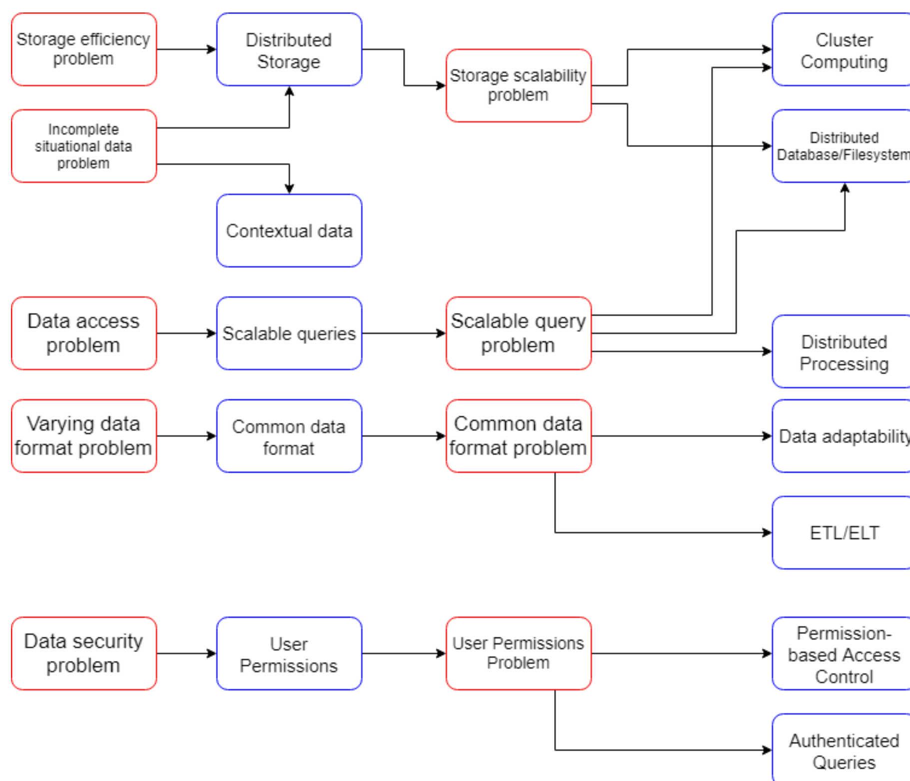

Figure 3.1: The synthesis diagram of the radar information system.

3.1 Synthesis overview

Figure 3.1 presents an overview of the synthesis that has been performed for this project. In the introduction to this chapter, we derived a few main technical problems from the requirements. These are the problems that need be solved by the solution of this project. These technical problems are denoted in the figure with a red box. The blue boxes all contain solution domains, from which abstractions may be found that solve the corresponding technical problem.

The storage efficiency problem needs to be solved by storing data in a way that it is at the very least somewhat structured. The current solution with large and cumbersome log files dose not work. A solution to this is to separate this log data and store it in a distributed manner. By doing this, a scalability problem is introduced. The solution to this problem in this project lies in the cluster computing and distributed storage solution domains. For the cluster computing solution domain, the choice was made to use technologies surrounding the Apache Hadoop software ecosystem. A justification for this choice may be found in 4.1. The storage part of the architecture and its specific functional and non-functional requirements will be described in Section 3.2.2.

The way in which data is loaded into the system will be discussed in Section 3.2.3. In the industry, this process is known as Extraction, Transformation and Load (ETL) or Extraction, Load and Transformation (ELT). In that section, the functional and non-functional requirements for such a subsystem will be described.

The system needs to be able to issue queries that retrieve data from the storage. There are various techniques available in the Apache Hadoop software ecosystem that can perform this. In Section 3.2.4, a look is taken at the functional and non-functional requirements of the system when it comes to running user-defined queries to retrieve data.

The data we wish to store is generated by different sources, and different types of sources. It is highly likely that there is a variance in the data formats that exist in this realm. Since the idea is to have a centralized system in which all data may be stored, there needs to be a way to correlate data that is related to each other, even when it is stored in a different format or unit. Functional and non-functional requirements that describe how adaptability should be implemented in the system are discussed in Section 3.2.5.

The data we will store needs to only be accessible by users with the correct permissions in the system, or function within the company. In order to enforce this, we will need an authentication mechanism that will only allow users access to (parts of) the data if they can identify themselves as a user capable of accessing the system, as well as being able to identify themselves as a user that is allowed to access

the data they are trying to gain access to. The authentication solution domain is discussed in section 3.2.6.

3.2 System architecture

From the synthesis process, a preliminary high-level architecture was created. This section describes the decomposition of that architecture, and gives a description of each individual part's responsibilities within the system.

3.2.1 Architecture decomposition

At a high level, we can distinguish between various components that make up the system as a whole. Figure 3.2 shows a high-level and abstract overview of the architecture that needs to be implemented.

Firstly, we have a *Source* that sends messages containing sensor data to the system. A *Source* can be any sensor in the context of this project. There can be many *Source*s that send data to the system at once, and the system needs to be able to distinguish between them.

Users of the system can create a Data Definition Specification (*DDS*) for each specific message type that is sent to the system by a *Source*. This specification contains a list of fields that is expected within the message, and it is used to validate inbound data. The specification can be created via a User Interface.

Next, we have an *ETL* component, which is responsible for handling incoming data from the various *Source*s that are sending messages to the system. One of the transformations from the *ETL* component is the validation of incoming data. Input data fields are compared to the *DDS* in order to judge data as valid or invalid. After data is deemed valid and completes any other transformation steps that may have been defined, the *ETL* process will forward the data to the *Storage* component. Invalid data will not be ingested into the *Storage* component and will be discarded.

Users can also use the user interface to define a *Relation Specification*. This specification contains relations between fields of different message types. When two fields have an Equality or Convertible relation to each other (See Requirement 6), they need to be listed in this specification in order to make a query's output uniform. In the case of a Convertible relation, an arithmetic conversion also needs to be applied to the data to guarantee output uniformity.

*End User*s can query the system in order to retrieve from *Storage*. During the querying process, the *Relation Specification* will be used by the *Query Engine* in order to correlate data to a common output.

The rest of the sections of this chapter will describe the main parts of the architecture:

- *Storage*
- *ETL*, including:
  - *DDS*
- *Query Engine*, including:
  - *Relation Specification*
  - *Query Output*

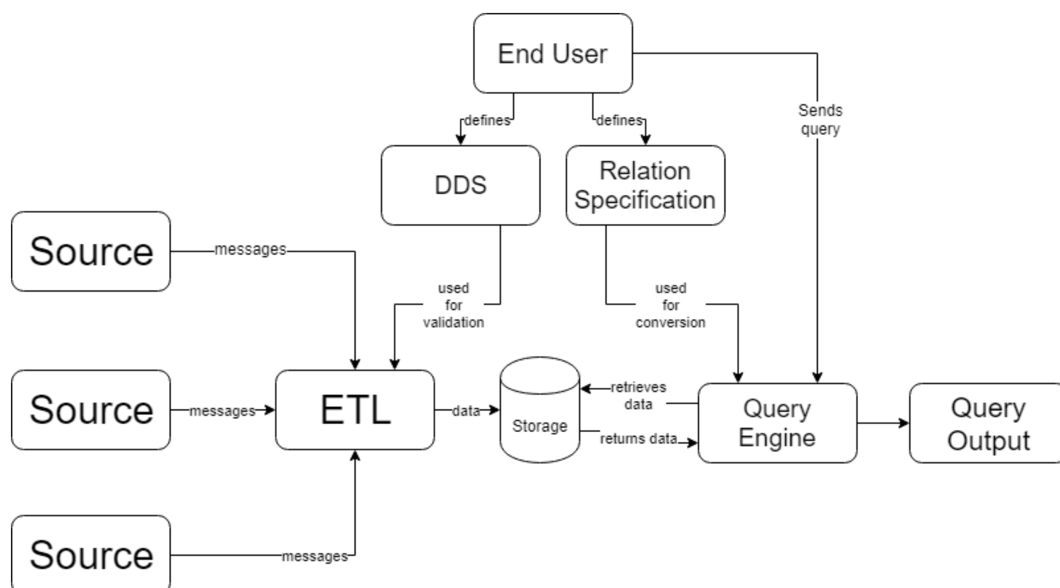

Figure 3.2: High-level architecture of the system.

3.2.2 Storage Mechanisms

In this section, the storage requirements will be decomposed further based on the results of the synthesis process. This decomposition consists of Functional and Non-Functional Requirements, each of which should be satisfied by the chosen solution. Based on the related technologies listed in Section 4.2, we will choose technologies that best satisfy the FRs and NFRs.

3.2.2.1 Functional Requirements

Table 3.1 lists the functional requirements for the storage system in order of importance. The section below the table will describe each functional requirement in greater detail.

FR#	Requirement
STO-FR1	The system must be capable of storing different types of sensor data.
STO-FR2	The system must be able to store data of which the format is not known at the time of system design.
STO-FR3	The system should be schema-less.
STO-FR4	It should be trivial to translate between the input data format (e.g. JSON, XML, binary log files) and the storage format used by the database.

Table 3.1: Storage Functional Requirements

STO-FR1: The system must be capable of storing different types of sensor data. As follows from the main requirements of this project, the system needs to be capable of storing sensor data. We have demonstrated that this data can consist of various types of messages per sensor data type, each type possibly varying in terms of granularity and size.

STO-FR2: The system must be able to store data of which the format is not known at the time of system design. We do not know the format of all radar data that will be stored in the system at the time of designing the system architecture. Therefore, the system needs to be capable of adapting and adding new data formats to its storage mechanism at any given point in time.

STO-FR3: The system should be schema-less. This requirement logically follows from FR2. Since we do not know the structure of the data beforehand, the ideal situation would be to have a system that is agnostic when it comes to the database schema. As per STO-FR2 and the requirements ETL-FR4 and ETL-FR5, data should still be validated before entering storage.

STO-FR4: It should be trivial to translate between JSON and the storage format used by the database. Radar data originally consists of differing data formats. An example of such a format would be a binary data file. Currently, the choice has been made to convert these files into a JSON data format before storing anything in the system.

3.2.2.2 Non-Functional Requirements

Table 3.2 describes non-functional requirements for storage systems in order of importance. The section below the table will describe each NFR in greater detail.

NFR#	Requirement
STO-NFR1	Completeness: Stored data should contain all components necessary to determine the state of a data source and the performed measurement.
STO-NFR2	Validity: Data entered into the system should always be valid.

Table 3.2: Storage Non-Functional Requirements

STO-NFR1: Completeness: Stored data should contain all components necessary to determine the state of a data source and the performed measurement. In order to effectively be able to use the data that is stored in the system, it needs to be complete. There needs to be a guarantee that all data that is ingested into the system will remain stored together, and will remain intact in its entirety.

STO-NFR2: Validity: Data entered into the system should always be valid. All data that is being stored should always conform to one of the designated Data Definition Specification (DDS) created by one of the end users. This behavior will be enforced by the ETL process as described in Section 3.2.3.

3.2.2.3 Architecture diagram

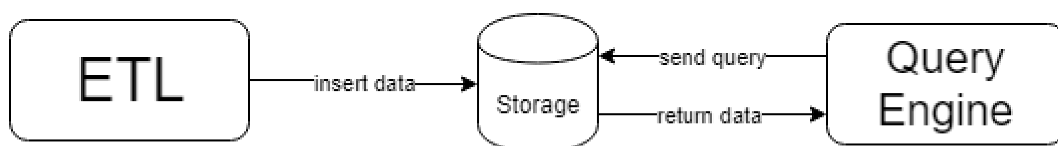

Figure 3.3: Storage architecture diagram

In Figure 3.3, a zoomed-in version of the architecture surrounding the envisioned storage system is given. It consists of the *Storage* itself, an *ETL* component and a *Query Engine* component.

As we have established during synthesis, the storage solution will need to be some kind of distributed database. This means that data will be stored on separate physical machines in a computer cluster.

The *ETL* component is the input for the storage system. It receives data from data sources and applies any necessary transformations. At the end of the ETL process, the data is persisted in the chosen storage solution.

3.2.2.4 Open questions

With big data being a relatively new field in terms of mainstream adoption, it is to be expected that there are still a lot of unanswered questions for each individual project where it is used. This section will highlight some of the most glaring and important problems in the big data solution domain. The answers to these questions will be given in 5.1.

Question: How can we ensure that we continue to be able to store and process data with linear scaling, given the exponential growth of data size versus the mere linear growth of processing power? One of the most important challenges that the field of big data faces at the moment, is the continuity of scalability. Currently, data sets across enterprises worldwide double in size every 1.2 years. [12] Meanwhile, Moore's Law states that the amount of transistors in integrated circuits doubles every two years. [13] This would mean that processing power would also double once every two years. This discrepancy will gradually lead to a larger scalability problem. Additionally, it is expected that Moore's law will not last forever, as we will eventually reach a physical limit, whereas the rate at which data grows seems to be exponential.

In our solution, which storage mechanism would prove beneficial to store data? There are several different ways to perform storage of big data. One one side, there are the traditional row-based relational database systems. On the other, there are column-oriented storage mechanisms that are seeing more adoption in the big data domain. An exploration of this particular column-based mechanism will be given in 4.2.

How compatible are big data storage and processing tools compared to each other? There are quite a few tools available in the field at the moment. Not all of these tools have equally well documented compatibility matrices when it comes to how well they mesh with existing tools that complement their functionality. It is important to research how compatible tools are in relation to each other, so that we

can choose the correct software suite as a solution to our problem. An exploration of storage mechanisms and tools can be found in 4.2.

3.2.3 Extract, Transform and Load (ETL)

In this section, we will look at how we will be loading data into the system. We will do so by defining both functional and non-functional requirements, and zooming in on the role that ETL plays in the overall architecture.

3.2.3.1 Functional Requirements

Table 3.3 lists the functional requirements for ETL technologies in order of importance. The section below the table will describe each FR in greater detail.

FR#	Requirement
ETL-FR1	The system should be able to ingest streaming data.
ETL-FR2	The system should be able to ingest batch data.
ETL-FR3	It should be possible to upload or place a data specification pertaining to specific radar data types (on)to the system.
ETL-FR4	The system should check incoming data against the relevant data specification.
ETL-FR5	The system should reject any data that does not match the specification.

Table 3.3: ETL Functional Requirements

ETL-FR1: The system should be able to ingest streaming data. Radars are complex systems that constantly spit out large amounts of data. In order to process all of this, the system should employ an ETL technology that is capable of handling large streams of data. Thales employees say that the incoming data rate can range from 1 megabit per second (Mbps) to 20 gigabits per second (Gbps).

ETL-FR2: The system should be able to ingest batch data. Sometimes, radar data may need to be read from older log files. In order to facilitate this, the system should be able to ingest data from these log files. There is a significant difference between batch and streaming data, so we should discuss them separately when looking for technologies that satisfy these two requirements. Currently, the amount of logged data from an average development stored on carriers consists of around 20 terabytes (TB, yet to be confirmed). In total, the amount of data stored across carriers at Thales is estimated to be hundreds of terabytes (TB).

ETL-FR3: It should be possible to upload or place a data specification pertaining to specific radar data types (on)to the system. In order to always have a "clean" data set when it comes to a particular type of radar, we will use a data specification that needs to be provided for each type of radar. This data specification may be uploaded in any way, as long as it can be used for data verification.

ETL-FR4: The system should check incoming data against the relevant data specification. Incoming data claiming to be from this particular radar type will be checked against the specification mentioned in ETL-FR3. Data that matches the format in the specification is considered as valid data, and will continue through the rest of the ETL process.

ETL-FR5: The system should reject any data that does not match the specification. Data that does not match the specification mentioned in ETL-FR3 should not be allowed into the system, and will throw an error.

3.2.3.2 Non-functional Requirements

Table 3.4 lists non-functional requirements for ETL technologies in order of importance. The section below the table will describe each NFR in greater detail.

NFR#	Requirement
ETL-NFR1	The system should be able to process data quickly, so that data does not have to wait in the queue for long to be ingested. (Performance in time)
ETL-NFR2	The system should be able to adapt to new data specifications being added, preferably on-the-fly. (Adaptability)
ETL-NFR3	The system should be easy to edit or develop for in the case anything changes about the ETL process. (Maintainability)

Table 3.4: ETL Non-Functional Requirements

ETL-NFR1: The system should be able to process data quickly, so that data does not have to wait in the queue for long to be ingested. In an ideal situation, data should never have to wait to be ingested into the system. Of course, this cannot be guaranteed, and as such we should try to keep the delay as low as possible. In streaming situations, incoming data can go from as low as 1 megabit per second (Mbps) to as high as 20 gigabits per second (Gbps), depending on sensor type and settings. Ideally, the data throughput in a final cluster would be close to the higher

bound. The further the actual throughput is from the higher bound, the more data will need to wait in queue in situations where the inbound data rate is at its highest.

ETL-NFR2: The system should be able to adapt to new data specifications being added, preferably on-the-fly. In reference to ETL-FR3, ETL-FR4 and ETL-FR5, adaptability can be measured by the ability of the system to use a newly added data specification immediately, and perform the activities listed in these requirements correctly.

ETL-NFR3: The system should be easy to edit or develop for in the case anything changes about the ETL process. Requirements of the ETL process are subject to change. In order to adapt the system to deal with these changes, the programs used to perform ETL should be as simple and modular as possible, so that any changes are easy to make.

3.2.3.3 Architecture diagram

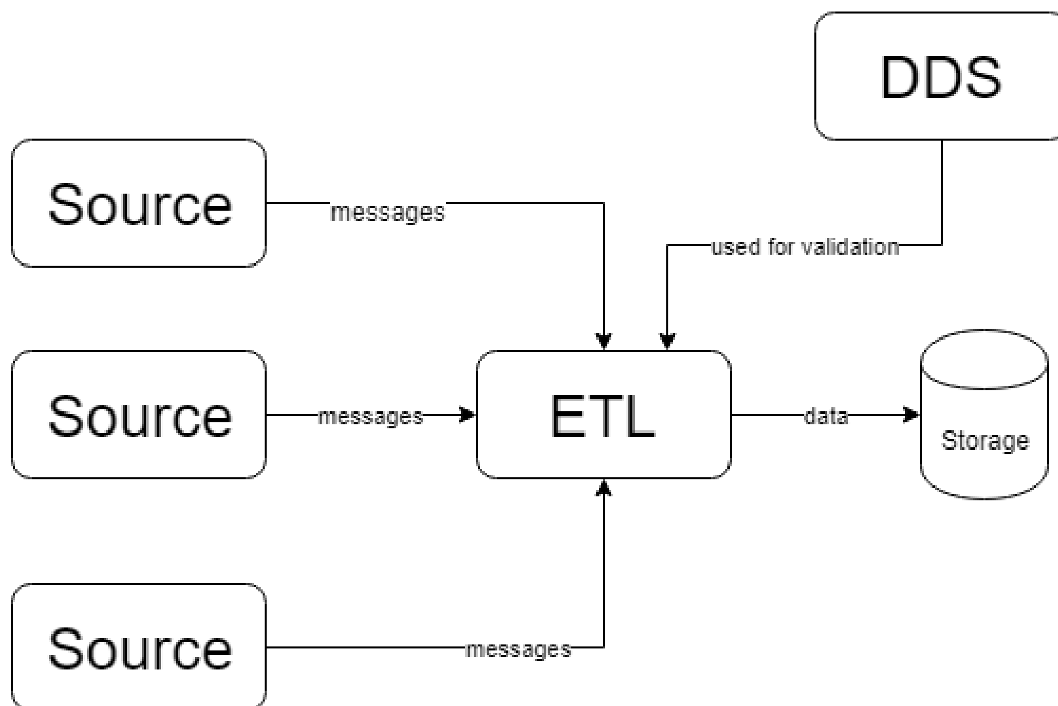

Figure 3.4: ETL architecture diagram

The system must be able to receive messages from multiple *Source*s. These messages are loaded into the *ETL* component, where they will undergo validation. Any data that is considered valid will be led through the rest of the Load process, and will be persisted in the *Storage* subsystem.

Validation happens with the help of a so-called Data Definition Specification (*DDS*). This specification contains a list of fields that must be contained by a specific message sent by a *Source*. If the message matches the specification, then it will pass validation. If it does not, it will be rejected.

3.2.4 Querying

In this section, we will start by discussing the individual requirements for querying the system. We will also take a brief look at the role that the querying system plays in the overall architecture.

3.2.4.1 Functional Requirements

Table 3.5 lists the functional requirements for ETL technologies in order of importance. The section below the table will describe each FR in greater detail.

FR#	Requirement
QUE-FR1	It should be possible for a user to create complex queries to extract data from the cluster.
QUE-FR2	A user should be able to query single data source types without any sort of conversion being needed.
QUE-FR3	A user should be able to query different data source types at once.
QUE-FR4	In the case where multiple data source types are queried, and some columns have a Convertible relation towards each other, they should be converted in order to get an Equality relation.
QUE-FR5	All related fields in the result set of a query should have Equality relations towards each other.

Table 3.5: Querying Functional Requirements

QUE-FR1: It should be possible for a user to create complex queries to extract data from the cluster. Data should be extractable from the cluster for various purposes. Users should be able to add filters to specific fields they are querying, in order to retrieve the resulting data set that they want to use.

QUE-FR2: A user should be able to query single data source types without any sort of conversion being needed. When retrieving data from a single data source, there is no need for data translation. All fields from the data source type shall be returned to the user in their original format.

QUE-FR3: A user should be able to query different data source types at once. A user should be able to query any number of different data types at once. As QUE-FR4 and QUE-FR5 describe, there will be some conversion needed.

QUE-FR4: In the case where multiple data source types are queried, and some columns have a Convertible relation towards each other, they should be converted in order to get an Equality relation. As has been mentioned, data source types may contain data that describe the same metric, but they may be stored in differing formats or units. In order to retrieve a uniform output from the database, a conversion will need to be applied to unify all the data.

QUE-FR5: All related fields in the result set of a query should have Equality relations towards each other. The resulting data set should be uniform, so the query should apply all relevant conversions.

3.2.4.2 Non-functional Requirements

Table 3.6 lists non-functional requirements for ETL technologies in order of importance. The section below the table will describe each NFR in greater detail.

NFR#	Requirement
QUE-NFR1	Performance in time: Queries issued by the user should complete as quickly and as efficiently as possible.

Table 3.6: Querying Non-Functional Requirements

QUE-NFR1: Performance in time: Queries issued by the user should complete as quickly and as efficiently as possible. In the case where different technologies satisfy all of the Functional Requirements, time performance will be used to decide on a technology to implement in the final system design. There is no real requirement as to how fast the query should be, but the technology that does it the fastest will be chosen.

3.2.4.3 Architecture diagram

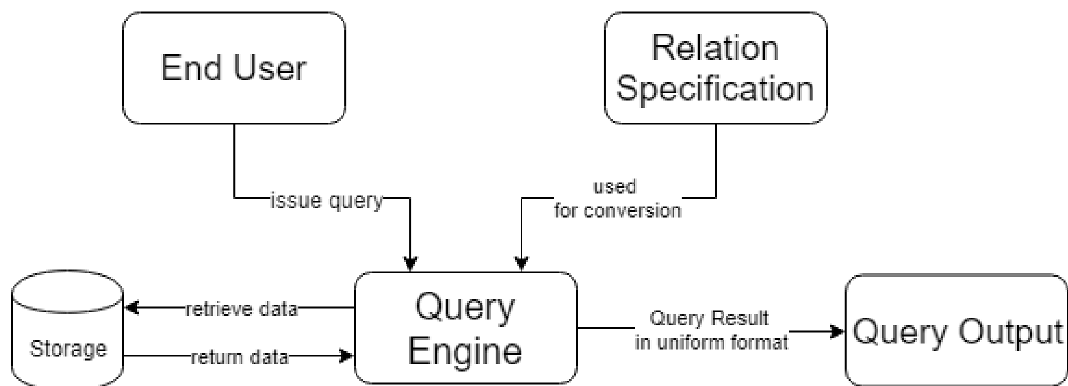

Figure 3.5: Querying architecture diagram

An *End User* of the system may issue a query through a user interface. In this interface, they will be offered a listing of all existing data fields, after which they can create a query, listing all data they would like to have, and any ranges that may apply to these fields. This query is then sent to the *Query Engine*.

The *Query Engine* retrieves data from the distributed *Storage*, and applies the query to the data set that is returned to it. Should there be any required conversions in the *Relation Specification* for any of the data fields that were queried, then the conversion formulas will be retrieved and applied to the data. After the query and conversions have completed, the result of the query will be returned to the *End User* as the *Query Output*.

3.2.5 Adaptability

This section will deal with the design aspects concerning the adaptability of the system. In the context of this project, adaptability is described as the ability of the system to deal with different sensor data types by having the ability to cross-reference them, and outputting them as if they were of the same format, should the user choose to query for them.

3.2.5.1 Functional Requirements

Table 3.7 lists the functional requirements for ETL technologies in order of importance. The section below the table will describe each FR in greater detail.

FR#	Requirement
ADA-FR1	End users should be able to define a base specification that all data source types can convert to.
ADA-FR2	It should be possible for end users of the system to define conversions between ambiguous fields and a relevant base specification.
ADA-FR3	Relations must contain a numerical conversion formula.
ADA-FR4	By applying conversion when multiple radar data types are queried, the resulting output should be in a uniform format.

Table 3.7: Adaptability Functional Requirements

ADA-FR1: End users should be able to define a base specification for radar data. This base specification is a format which radar data formats can convert to freely. When a multitude of these formats are queried from the database, relevant conversions will be applied to return a uniform output.

ADA-FR2: It should be possible for end users of the system to define conversions between ambiguous fields and the base specification. Conversions from relevant fields to the base specification must be defined by users. Whenever a new radar data format is stored in the database, these conversions have to be created in order to keep output data uniform.

ADA-FR3: Relations must contain a numerical conversion formula. Conversions between relevant fields and the base specification may be considered as strictly numerical for the purpose of this project.

ADA-FR4: By applying conversion when multiple radar data types are queried, the resulting output should be in a uniform format. It is the end user's responsibility to make sure all conversions are defined, complete and accurate. If this is the case, then all output will be in the correct format.

3.2.5.2 Non-functional Requirements

Table 3.8 lists non-functional requirements for ETL technologies in order of importance. The section below the table will describe each NFR in greater detail.

NFR#	Requirement
ADA-NFR1	Performance impact: The influence of the technology used to store data and cross-reference specifications on performance should be as low as possible.

Table 3.8: Adaptability Non-Functional Requirements

ADA-NFR1: Performance impact: The influence of the technology used to store data and cross-reference specifications on performance should be as low as possible. In case multiple techniques come up to perform this adaptability, the way with the lowest impact on performance will be chosen as the solution to use in the final design.

3.2.5.3 Architecture diagram

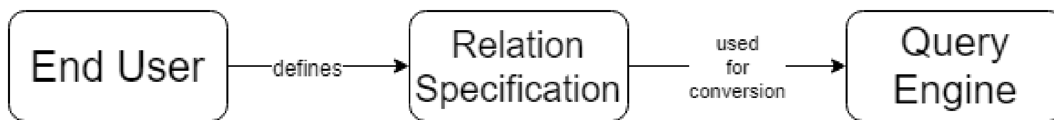

Figure 3.6: Adaptability architecture diagram

An *End User* of the system may create a base specification that will be used as the query output format. They may also create relations between fields of a radar data type and the base specification they have defined. Relations that apply to a single radar data type put together are called a *Relation Specification*. Each data format has its own *Relation Specification* that contains numerical conversions to achieve a uniform query output.

3.2.6 Access Control

In this section, we will look at ways in which access control mechanisms could be implemented in the final solution. We will do so by defining Functional and Non-Functional requirements for this particular subsystem.

3.2.6.1 Functional Requirements

Table 3.9 lists the functional requirements for ETL technologies in order of importance. The section below the table will describe each FR in greater detail.

FR#	Requirement
ACC-FR1	It should be possible to restrict access to (certain parts of the) data on the cluster.
ACC-FR2	It should be possible to give (a certain subset of the) end users access to the data, when they provide the correct credentials.
ACC-FR3	It should be possible to force specific data to be encrypted (should happen during ETL).

Table 3.9: Access Control Functional Requirements

ACC-FR1: It should be possible to restrict access to (certain parts of the) data on the cluster. End users of the system may not have permission to access certain parts of the overall data set. This needs to be reflected in the system. Administrators should have the ability to revoke access to subsets of data for each user.

ACC-FR2: It should be possible to give (a certain subset of the) end users access to the data, when they provide the correct credentials. Whenever users want to access data, they should have the correct permissions to do so. Administrators should have the ability to grant permissions to users to access specific subsets of the data.

ACC-FR3: It should be possible to force specific data to be encrypted (should happen during ETL). It should be possible for administrators to force encryption on specific subsets of the data. Users should have the correct permissions in order to retrieve the original data set in the case where encrypted data is queried.

3.2.6.2 Non-functional Requirements

Table 3.10 lists non-functional requirements for ETL technologies in order of importance. The section below the table will describe each NFR in greater detail.

NFR#	Requirement
ACC-NFR1	Performance in time: The access control mechanism that is used should have as little effect on overall ETL and query performance as possible.

Table 3.10: Access Control Non-Functional Requirements

ACC-NFR1: Performance in time: The access control mechanism that is used should have as little effect on overall ETL and query performance as possible. If there are multiple technologies that satisfy all Functional Requirements, performance should be the deciding factor to choose which technology should be used. There is no requirement that states ideally how little the influence of such a technology should be, so the technology that has the least influence will be chosen.

3.3 Project objectives

After the literature research, we are left with a few open questions. The answers to these questions will need to come forth from new insights that will come to light during experimentation and the execution of the project. This section provides a summary of the questions that will be answered over the course of the project.

1. Which software is most suitable for the storage of the data that will be used, based on the following criteria?
   (a) How well does the storage software scale when the data set continues to increase in size? What are the techniques available to improve storage size scalability?
   (b) How quickly can data be persisted on the chosen system? What techniques are available to further improve the speed at which it is possible to store data?
   (c) How compatible is the chosen storage software with related data processing tools?
2. Which software is most suitable for the processing of the data that will be used, based on the following criteria?
   (a) How well does the processing software scale when the data set to be processed increases in size? What are the techniques available to improve processing size scalability?
   (b) How quickly is the system able to process data with the chosen processing tools? What techniques are available to further improve the speed at which it is possible to process data?
   (c) How compatible is the chosen data processing tool with the chosen storage layer?
   (d) How easy is it to adapt the chosen data processing tool to applications that will use the results of the processed data set?
3. Which security concepts are best applicable to the system under development, and how should they be implemented?
   (a) How can user permissions be implemented in order to guarantee correct access control of (subsets of) big data?
   (b) How well does the application of such permissions scale when it is done during the ETL process?

4. How can we incorporate varying source data formats into the system, while promoting internal uniformity of data?
   (a) How can we define a uniform destination data format in order to be able to map all source radar data to it? This implies that the destination format is a superset of all field types from every radar data type.
   (b) How would already existing mappings interact with changes made to the uniform data format?
   (c) What is the best way to allow users to create definitions of source data formats, while linking them to a uniform destination data format via mapping functions?
   (d) What is the best way to define data translations?
   (e) Is there a way to let new source data formats derive from existing ones, and how can this be interfaced towards the user?
5. How can we structure the ETL process for a big data architecture?
   (a) What are the best techniques available for performing ETL on top of the previously chosen tools?

Chapter 4

Existing technologies

In this chapter, an insight will be given into the existing technologies within the Apache Hadoop ecosystem, more precisely the ones that are relevant to the requirements set for the project, and the ones that satisfy one of the roles of the system mentioned in the architecture description in Section 3.2.

4.1 Big Data in General

Big data is a field that is relatively new, and thus there are many challenges that experts in the field are attempting to overcome. This section lists a few of the challenges and problems that are most common in big data from the viewpoint of the Apache Hadoop software ecosystem.

4.1.1 Big data challenges

CPU-heavy, I/O-poor. CPU performance capacity follows a trend in which it doubles every 18 months. This is commonly known as Moore's law. Information, however, grows at an exponential rate, while processing techniques grow relatively slowly. In current Big Data applications, most of these techniques cannot quite solve all of our problems yet, especially in fully real-time applications. [14]

Data inconsistency. Inconsistencies will always find their way into any large dataset. They may occur at any level of data granularity, of which there are five [15]:

1. Data
2. Information
3. Knowledge
4. Metaknowledge

5. Expertise

These inconsistencies may have a negative impact on big data analysis methods, and may even adversely affect the results of such analysis. The consistency of data is also referred to as veracity. [16]

Scalability. Scalability can be subdivided into two subproblems: Data storage scalability and data processing scalability. Data storage scalability depends on how much data we are able to store in our big data cluster, whereas data processing scalability depends on how much data we are able to process in parallel, and how much data we can process within a timely fashion.

The scalability problem is the main problem that the big data solution domain is trying to solve. Data that amasses over a longer period of time usually transcends the constraints of traditional RDBMS running on singular systems. Since these systems do not run in a distributed fashion, it is not possible to solve the scalability problem simply by adding more machines. The only option would be to add more (powerful) hardware to the single machine, which would still leave it severely limited, as far more processing power may be achieved by running queries in parallel across multiple machines compared to a single, more powerful machine. A single machine will quickly run out of storage space, processing power capacity and/or memory.

Timeliness. Timeliness can be divided into two separate components:

- The time between the expectation of gaining access to data, and the time where access actually occurs.
- The time between collection and processing of data until the time where the processed data is ready to be used.

Data security. Providing security for data on a large scale is a challenge. For the purpose of this assignment, data security will be defined as ”granting data access to those who have the correct permissions, and denying access to those who do not, as well as making sure that unwanted intrusions are prevented.”. Therefore, the keywords for security in this regard are authentication, authorization and intrusion prevention.

Storage limit. We can only store so much data as there is collective space available.

Processing limit. Although distributed processing is generally in place in these systems, the maximum processing speed is limited by the collective capacity of the systems.

4.1.2 Hadoop KSPs

Apache Hadoop is a software framework that provides opportunities to store large amounts of data in a distributed fashion. By harnessing the collective processing capabilities of a set of computers rather than just one, there are several things that Hadoop aims to offer.

Scalability. Many data-producing constructs nowadays produce data that is too large for singular computer systems to handle. In response to this new trend, big data systems were invented. By using multiple computer systems and networking them together with the same purpose, it is possible to introduce scalable systems that can handle large amounts of data.

Innovation. Big data technologies can replace a lot of today's highly-customized technology by using standard solutions that are able to run on commodity hardware.

Accessibility. Many big data technologies are open-source, which makes it cheaper to implement software solutions by using them compared to traditional proprietary technology.

Reliability. Hadoop guarantees data redundancy by replicating data several times across a cluster. This option is configurable by the system administrator. By providing this form of data reliability, systems that are resilient against potential failures of single nodes can be built. Data will continue to be available even in the case of such a failure.

4.1.3 Apache Hadoop

This section will describe Apache Hadoop's core components and how they function.

4.1.3.1 Master-Slave architecture

Big data architectures often build on the master-slave principle. In this principle, all the machines in a cluster are interconnected and each have one of two specific roles. A select few of the machines will become master nodes, which will coordinate and run the top-level applications. The others will become slave nodes, which are responsible for the storage and processing of data.

An example of how the master-slave architecture is employed in Apache Hadoop may be found in Figure 4.1. Note how Hadoop distinguishes between master and slaves by using its own terms: The NameNode is the master node, while slave nodes are referred to as DataNodes. For clarity, we will explain the significance and the function of both types of nodes.

As shown, a master node consists of a few components that are able to perform a certain set of actions to manage the cluster. Firstly, a master node contains

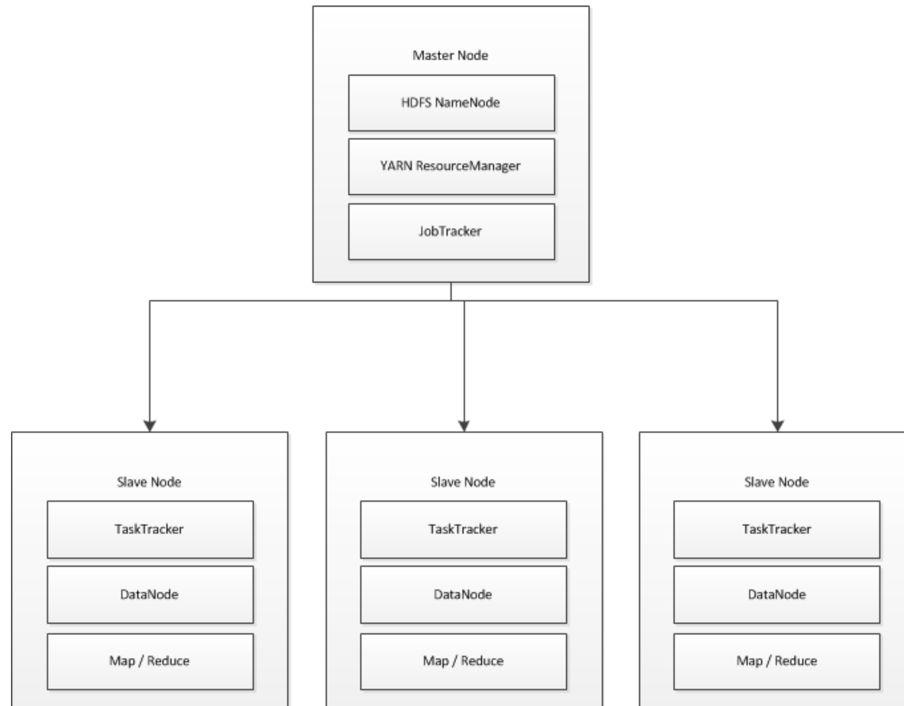

Figure 4.1: A master-slave architecture as it is used in Apache Hadoop.

the HDFS NameNode component, which provides management options over the Hadoop file system. It keeps a directory tree of all files stored on HDFS, and tracks which machines contain specific files. The NameNode does not store any of the data on itself. Client applications may ask the NameNode: [17]

- For the location of a file.
- To make certain additions to a file.
- To copy a file.
- To move a file.
- To delete a file.

The second component on a Hadoop master node is the YARN ResourceManager. YARN is a resource negotiator, which means that it manages the resources that are available in the cluster. Therefore, it also indirectly helps manage the distributed applications that are being run on the cluster. In order to perform this task, it requires the help of the NodeManagers that are run on each slave node, as well as ApplicationMasters, which are run for each application that is currently being executed on the cluster. [18]

The last component of a master node is the JobTracker. It sends out MapReduce tasks to specific cluster nodes. Ideally, these nodes will be the ones that already

have the data they need, such that data transferring is minimized as much as possible. To summarize, these are the functionalities that are offered by the Hadoop JobTracker: [19]

1. Clients submit jobs to the JobTracker.
2. The JobTracker requests the location of the data needed for the job from the HDFS NameNode.
3. The JobTracker searches for nodes that have available slots, while also containing the location of the data, or being close to a node that does. It does so by querying data nodes' TaskTrackers.
4. The JobTracker will then submit the work to available and eligible TaskTracker nodes.
5. The TaskTrackers will be monitored for the duration of the job. They will periodically send out heartbeat messages to signal that they are still actively working on the job. If these heartbeat messages stop, the job is assumed to have failed. This work will then be sent to a different TaskTracker node.
6. Additionally, a TaskTracker will also notify the JobTracker whenever a job fails to execute. The JobTracker will then be able to decide further action: it may move the work to a different node, it may mark the data record that created the error as an errant record. It may even mark the TaskTracker node as an unreliable node, disallowing it from being used for further jobs.
7. When the work is completed by the TaskTracker, the JobTracker will update its status.
8. Client applications may query the JobTracker for job progress information at any time.

From the described components, we can clearly formulate the mission that a master node has in a cluster: To keep a log of data storage locations, to negotiate resources with the rest of the cluster for job resource allocation, and to schedule the jobs that are executed across the cluster.

Next, let's look at the components of the slave node. As we have established, it consists of the TaskTracker, DataNode and MapReduce components. Some of these have already been mentioned a few times during the master node descriptions, and we will go into more detail about them below.

As the name suggests, a TaskTracker is a node in the cluster that accepts tasks from a JobTracker. These tasks can be Map, Reduce and Shuffle tasks; these will

be described in the MapReduce section. Each TaskTracker node has a set number of slots, indicating the number of jobs it is able to accept at a time. Whenever a JobTracker attempts to find a node that is capable of performing a certain job, it will first look on the DataNodes that host the data needed by the job. If any of these nodes have a free slot, the job will be scheduled on it. If it does not, then it will attempt to find a node that is physically close to the data and see if any of those nodes have any free slots.

Once a job has been submitted to a TaskTracker node, it will spawn separate Java Virtual Machines (JVMs) to perform the required work. The reason for this is to make sure that a job failure does not bring the entire TaskTracker process down with it. These processes are then monitored by the TaskTracker. It will notify the JobTracker whenever work is completed, or when a failure occurs. As mentioned before, the TaskTracker will also occasionally notify the JobTracker that it is still alive via heartbeat messages, indicating that it is still active and working on the submitted job. These messages also contain information about the currently available amount of slots, keeping the JobTracker updated about potential locations where future work could be scheduled. [20]

A DataNode is a node that stores data as a part of HDFS. A functional filesystem will contain multiple DataNodes, with data being replicated across several nodes to provide data resilience. A DataNode connects to a NameNode on startup, and will then listen for requests for file I/O from the NameNode. Client applications will be able to communicate directly with a DataNode once the NameNode has given them the location of the data. MapReduce operations that have been sent to a node near the target data can also communicate directly with DataNodes in order to access file data. Data nodes are also able to communicate among each other in order to replicate stored data.[21]

4.1.3.2 Hadoop processing layer: MapReduce

MapReduce is a software framework that serves as the computation layer of Apache Hadoop. MapReduce jobs are divided into two (obviously named) parts. The “Map” function divides a query into multiple parts and processes data at the node level. The “Reduce” function aggregates the results of the “Map” function to determine the “answer” to the query. [22] It is shipped as a part of Hadoop Core. In this section, an insight will be given into the workings of MapReduce.

Distributed processing in Hadoop clusters can be performed by the implementation of Google’s MapReduce algorithm. This algorithm consists of several steps that allow data to be distributed among the different nodes in the cluster. These nodes will then individually process the data and forward their results. At the end, the result

set will contain the combined work of each individual worker node.

An overview of the MapReduce algorithm may be found in Figure 4.2. The MapReduce algorithm works as follows:

1. The MapReduce library will first split all input data files into chunks that are between 16 and 64 MB in size. The size of these chunks can be controlled via a user parameter. Then, it will start up instances of the MapReduce program on slave systems.

2. One of these instances will be deemed the master instance. This specific instance's job is to assign work to the slave instances. The master instance will pick idle worker nodes and assign them work to do. In total, the amount of map tasks is equal to the amount of chunks that were created in step 1, and an unspecified amount of reduce tasks.

3. A worker parses its input data and passes key-value pairs through to the mapper UDF.

4. The output key-value pairs of the mapper function are written to disk and are partitioned into regions by the partitioning function (sometimes known as ”Shuffle and Sort”). These regions are then passed back to the master so that they may be assigned to a reducer.

5. Once a reducer knows the location of the data, it will start sending RPC messages to the machine associatead with the location. Once the reducer completes reading data, it will sort all of the input key-value pairs by their key.

6. For each of the key-value pairs in the sorted set, the reducer receives it as input for its reduce function. The output of the reducer is written to an output file.

7. Once all mappers and reducers have completed their work, the job is done, and the master will notify the user application.

As the first step of a MapReduce job, the master will split the input data-set into independent chunks. These chunks are then divided across multiple map tasks, which process the data chunks in parallel. Between the map and reduce phase, the outputs of the map phase are shuffled and sorted. The reduce phase aggregates the results from the map phase. Finally, the resulting data-set is the output of the program. This output is usually persisted on the file system.

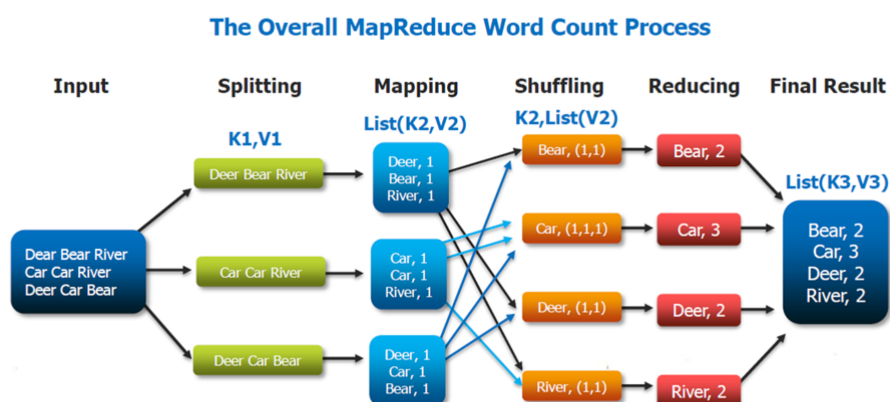

Figure 4.2: An overview of the inner workings of a MapReduce job. [23]

4.2 Storage mechanisms

Distributed storage in Hadoop clusters can consist of several different techniques. We will list some of them in this section.

First and foremost, Hadoop comes with its own storage layer implementation, which is the Hadoop Distributed File System, or HDFS for short. It is a file system that is able to read all of the slave nodes' storage in order to construct a complete storage overview. The end user can query HDFS on the master node and view all data stored on the cluster. HDFS also offers some handy configuration options, such as the option to replicate data several times and store the copies on different machines on the cluster.

Another technique which is commonly used in big data architectures is a NoSQL database. NoSQL stands for "Not Only SQL" or "Non-relational". NoSQL databases typically have six key features: Horizontal scalability, replication, a simple protocol, a weaker concurrency model than SQL's ACID (Atomic, Consistent, Isolated, Durable) transactions, efficient indexing and RAM usage and dynamic adding of data records. [24]

There's also a difference in optimization when it comes to storage tools. Some of these tools are better optimized for batch processing, such as HDFS [25] and Apache Hive [26]. These two are specialized in handling large batches of data, whereas they do not deal as well with real-time data. On the other hand, we have tools such as Apache Cassandra [27] and Apache HBase [28], which do support real-time data processing.

There is a distinction within Database Management Systems between the ways in which data is stored. Some data is stored with a row orientation, in which the key

RowId	EmpId	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	55000

Table 4.1: Sample relational table [29]

of the row precedes the rest of the record. To illustrate this, we will use an example. Let us suppose that we have the following traditional relational table:

Commonly, tables would be stored using serialization. Below, we can see an example of how the table would be serialized in a row-oriented system. The RowId is assigned to the system at the time data is stored, and is used as an internal pointer to a data record. Row-oriented systems are ideal if you need as much information from a row as possible. Examples of systems that would use these systems well are mostly object-oriented systems, where you want as much information about a particular entity in the system as possible. However, it proves to be somewhat inefficient in the case where you just want data from one, or a few particular fields.

```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```

Listing 4.1: Serialized relational table data[29]

To improve the performance of such field-specific queries, columnar data stores have been invented. Columnar data stores, as the name implies, are column-oriented. What this means for their structure is that data records are not stored in rows. Rather, the rows consist of all entries of a certain column. [30] If we were to serialize the above table in columnar format, it would look like this:

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

Listing 4.2: Serialized columnar table data [29]

4.3 Adaptability

In this section, we will briefly take a look at possible ways in which adaptability can be implemented in this system. As per the requirements, we will define adaptability as

the ability to validate incoming data according to a data format specification, and the ability to query data from different formats through a cross-reference specification.

The latter of these will consist of relations between related columns from different formats, and will define what type of relationship exists between the two. The relationship can be equality, in which case all data from either format will simply be output to the user who issued the query. However, this relationship can also require some kind of conversion, in which case that will need to be applied to the data in order to convert all data to the desired output format.

A solution for the adaptability problem could be to apply data unification. This technique defines a common standard for data in a specific field. An example of this can be found in the work of Teeters et al., who have defined a uniform data format for neurological data. In their work, they distinguish between two different challenges for data unification: Defining the data types and all the relationships between the different types, and giving users the option to store everything in their format. By building a system that would facilitate both of these challenges, they were able to define NeuroView, a compatible data format that could encompass neurological data as a whole.[31]

There exist several technologies for the conversion of data, some of them dating back as far as forty years. In 1975, Shu et al. came up with CONVERT, a high-level translation definition language for data conversion. This language gave users a visual representation of their defined translations, making the overall conversion work a lot more clear. [32]

El-Sappagh et al. have defined data conversion as a part of Enterprise Application Integration (EAI) in their work. EAI consists of some principles that allows for the integration of a set of systems. Data conversion is one of the steps that falls under the category of application integration. In many enterprises, data unification and conversion tools exist in proprietary integration kits. [33]

The common consensus about a way to handle the actual conversion of data seems to lie in the field of rule-based conversion. In this method, the source schema of the data to be converted is compared to the target schema. Before this comparison, the system already has knowledge of the rules needed for conversion between the source and target system. An example of such a conversion scheme is YAT, which is based on named trees with ordered and labeled nodes. Its language, named YATL, is declarative and rule-based. [34] Milo and Zohar propose a way to utilize schema matching in order to simplify heterogeneous data translation. In their implementation, they use the similarities between existing schemas to limit the amount of programming and configuration needed to make the conversion work. [35]

Storing data specifications In order to make the system adaptable to the use of new types of radar data, the system needs to know what kinds of data formats will be entering the system. Therefore, there needs to be a way in which users of the system can specify these data formats. After uploading such a specification onto the system, it can then be used to recognize and verify incoming data.

If the choice is made to use a NoSQL database to store data, then data can be stored without having to worry much about the structure of the data. These data models can be flexible in the sense that there is no need for the database management system itself to know which columns and fields are stored. Conversely, since there is no way to have a fixed table structure, information about the fields of each data format needs to be stored elsewhere. This is necessary in order to satisfy the functional requirements of the ETL process, which aim to guarantee data purity.

There are a few ways in which such a specification could be stored. One of the more obvious ones is to use the technologies that are already deployed on the system. This would consist of using a technology related to Apache Hadoop to store the data specifications.

Firstly, consider a scenario where data definition specifications are stored in native Hadoop, more specifically HDFS. This would entail writing down a specification in a pre-determined format, storing it in a file, and then storing that file in a specific location on HDFS. The ETL framework could then read this specification, keep it in memory as the Storm topology remains active, use it to validate incoming data, and route that to the correct table.

It would also be a possibility to simply store these specifications in a NoSQL database, along with the radar data, while keeping the specifications in a separate table. The technology used for the ETL process would then need to query this table to validate the incoming data against the specification.

Secondly, there is potentially a way to store specifications in a separate, small-scale SQL database. This database would then need to be queried in order to check incoming data against the specification.

Storing cross-reference specifications Some columns between different radar data formats may be related to each other, but may not carry the same notation. A user may wish to query data from different formats at once and wishes to have a single, clear overview of the information contained in it. Since data is stored in the system with its original format intact, it should be possible to store a specification that lists all columns from all formats that contain the same kind of information.

Because these columns can contain information in different units or notations, it should be possible to convert all of these measurements into the output of a single format. Therefore, each relation between columns should specify whether the infor-

mation is represented in the same way, and if it is not, then they should both specify how to convert their data to the other format.

Such a cross-reference specification will need to be stored on the cluster. Ideally, we would store them in the same way we store the data specification, in order to minimize dependencies on storage systems, and in order to minimize the total amount of storage systems we use for the solution as a whole.

4.4 Cryptography

Since the requirements state that we need to be able to assign permissions to users, as well as encrypt all data with an appropriate access policy, we will need to use Role-Based Access Control (RBAC), sometimes known as Permission-Based Access Control. In this paradigm, permissions are associated with roles, and users are added to their appropriate roles in the system. As a result, users will be able to gain access to data that requires the permissions they possess. [36]

Within RBAC, we can distinguish between two types of access control: mandatory access control (MAC) and discretionary access control (DAC). MAC policies are decided by a security administrator, and users may not make any modifications to grant access to files. The security administrator assigns permissions to a user, and that user may then only access files that require those permissions. DAC policies, on the other hand, allow users to define their own access control policies. As a result, they can decide who has access to files under their ownership.

Fairly recently, a new form of RBAC has come into existence. It is called Attribute-based Encryption (ABE). ABE is considered. to be a part of functional encryption (FE). As the name implies, it relies on the encryption and decryption of data depending on user attributes. These attributes are somewhat like permissions in the sense that every user has their own set of attributes which define their access to specific parts of the system.

ABE consists of two techniques called Ciphertext-Policy Attribute-Based Encryption (CP-ABE) and Key-Policy Attribute-Based Encryption (KP-ABE). The differences between these two techniques are the locations of the access policy and the attributes. In CP-ABE, the access policy is what is used to encrypt the data, and the attributes are contained within a user's secret key. In KP-ABE, it is the other way around: The user's secret key contains the access policy which lets them decrypt files with matching attributes. One of the big advantages of either of these techniques is that they are collusion-resistant schemes. The example below will illustrate how this is achieved in CP-ABE. [37]

Let's suppose we have the following scenario:

- We have a superset of attributes $\{A, B, C, D\}$.
- Let there be a file encrypted with policy $(A \wedge C) \vee D$.
- We have two users with their own secret keys:
  - User 1, with attributes $\{A,B\}$.
  - User 2, with attribute $\{D\}$.

With the above information, we know that a user that wishes to decrypt the encrypted file need to have either the combination of A and C, or the individual attribute D (or both) as their secret key's attributes. If they do not satisfy this requirement, they will not be able to decrypt this file.

In figure 4.3, we can see how what the overall structure of the CP-ABE system looks like in a Hadoop architecture: The Hadoop Distributed File System (HDFS) stores files and encrypts them. Users receive a secret key from the system which lets them decrypt files. Users will have to extract files from HDFS. In addition to this, they can only read the original file content if they have the correct permissions.

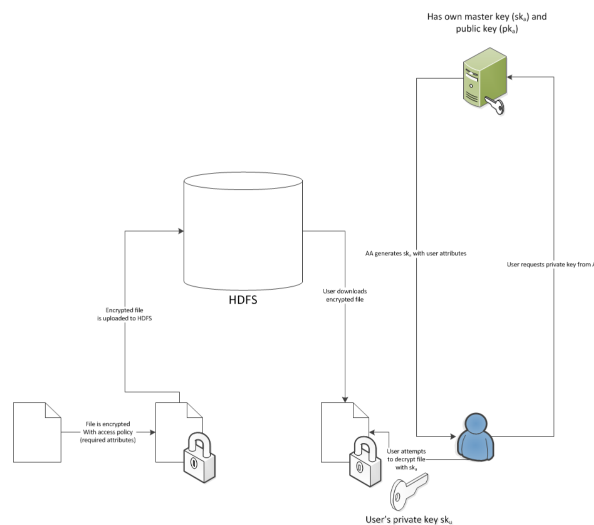

Figure 4.3: An overview of the structure within CP-ABE.

We can see what the process of CP-ABE looks like in figure 4.4:

- The Trusted Authority (TA) is responsible for generating secret keys for users. It also has its own public key and private key. The public key of the TA is also referred to as the master key.
- The encrypting party has a file they would like to store. This file needs to be encrypted with the public key of the TA, along with the access policy that needs to be enforced on the file. The encrypted file is then persisted into HDFS.

- The decrypting party retrieves the encrypted file from HDFS and attempts to decrypt it with its own private key. If they have the correct attributes to match the file's access policy, then the decryption will succeed, and the user will be able to read the file. If they do not have the correct permissions to match the file's policy, then decryption will fail, and the user will not be able to read the file.

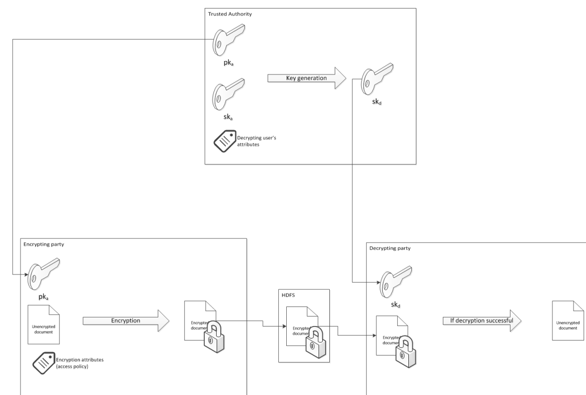

Figure 4.4: An overview of the process within CP-ABE.

4.5 ETL and ELT

As mentioned in section 4.1, big data technologies have come into existence to solve scalability problems. These technologies have become sophisticated enough to make hardware constantly scalable, simply by adding more hardware, we will be able to handle a linearly equal amount of data. Even though the problem is mostly solved by big data technology, it's still worth mentioning here, as it applies just as much to the retrieval and storage of data as it does to the parallel processing of it.

An interesting problem, however, is the heterogeneity of data. Essentially, this property means that data comes from different sources and therefore usually consist of different formats. However, this data does need to be connected to each other in order to process it properly. So how can we unify the data in such a way that it will be programmatically relatable to each other?

For this purpose, the concept of Extract, Transform and Load (ETL) has been created. This methodology for retrieving and storing data consists of three separate phases: The retrieval of data from many different data sources (Extraction), the application of transformation steps to the data (Transformation) and persisting the resulting information into a data store (Load), so that it may be processed further.

Another closely related paradigm is called ELT, where transformations are applied after the data has been loaded into the system, for example during processing. For the sake of not repeating, the rest of the text will refer to these concepts as ETL.

Within big data, there are a multitude of (combinations of) technologies that allow ETL to be implemented into the system. In order to decide which of the tools should be used for our solution, we need to weigh several criteria against each other. The criteria that will be selected on will be explained in Chapter 6, in which a few experiments will be conducted that compare these criteria across several ETL technologies in the context of this project.

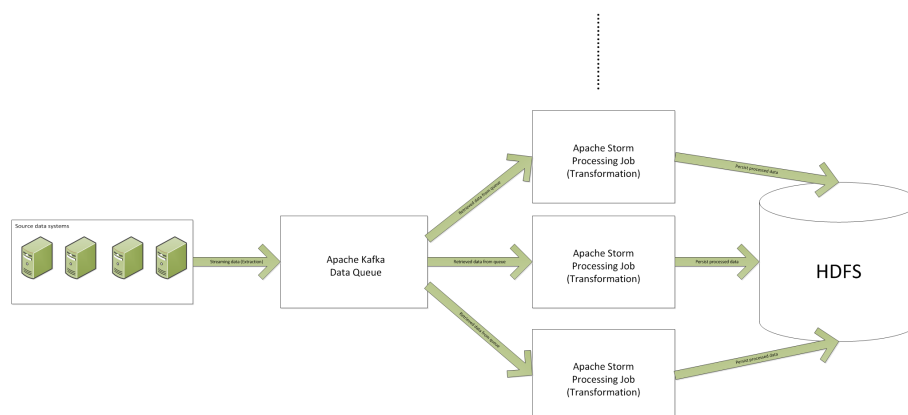

Figure 4.5: An example of ETL with Apache Kafka and Apache Storm.

Let us go over each of the individual stages of the ETL process, starting with Extraction. Out of the processes of ETL, it is probably the most important one, because the retrieval of data from source systems lays a foundation for the transformation and load processes. The Extraction process can consist of various techniques that may be used to retrieve data, such as (but not limited to) persistent file storages, databases and webpage scraping.

The Transformation stage consists of a set of rules or functions that run over the data set and modify the structure or contents of the data where necessary. Some parts of incoming data may not require any rules or functions to be applied to it. This data is known as pass-through data. Below are some examples of the potential filters that may be applied during this stage:

- Selectively loading columns, i.e. leaving other data out.
- Translating values
- Deriving a new value through a pre-defined formula
- Sorting
- Merging data

- Aggregating records
- Pivoting table (row-to-column or column-to-row, which might be useful when migrating data from a row-based to a columnar data store)
- Validation

This section describes the most notable technologies within the Apache software ecosystem that may be used for ETL purposes. For each software project, we will provide various statistics that will aid in reasoning which software is most suitable for this project. These statistics include the history, characteristics and overall quality of the software involved. In particular, we will be looking at Apache Storm, Kafka, Sqoop, Flume, Hive and Spark. This section will differ from the literature research in the regard that it will also highlight what the exact application of these technologies in this project would be.

4.5.1 Apache Storm

On its website, Apache Storm describes itself as a realtime computation framework for streaming processing. This means that Storm is capable of receiving data that is entering the system at a high volume or velocity, and processing it immediately.

Because of the fact that this description is rather open-ended, there are many different applications imaginable for Apache Storm. One of those applications is ETL or ELT. The capabilities of Storm make it an excellent candidate for ingesting streaming data into a storage cluster.

Storm employs a so-called ”topology” as a means of defining its architecture. A topology is a collection of data spouts and bolts, organized in a Directed Acyclic Graph (DAG) fashion. Spouts are the source of data streams that are processed by the rest of the Storm topology. They can be used to read data from an external data source (e.g. an Apache Kafka data queue) and propagate this data into the appropriate section of the topology.

Spouts have the capability to be configured as being reliable or unreliable. When configured as reliable, they can replay any tuple that has not yet been processed by the rest of the Storm topology. An unreliable spout does not re-emit any information after it has been emitted once already. Spouts are capable of outputting more than one stream, which turns the topology into a DAG where data will flow through based on pre-defined ”pathways”.

These pathways consist of so-called bolts. These little parts of the topology are responsible for processing incoming information. They can perform many different operations, such as filtering, validation and persisting data to databases. Just like spouts, bolts are capable of outputting multiple streams.

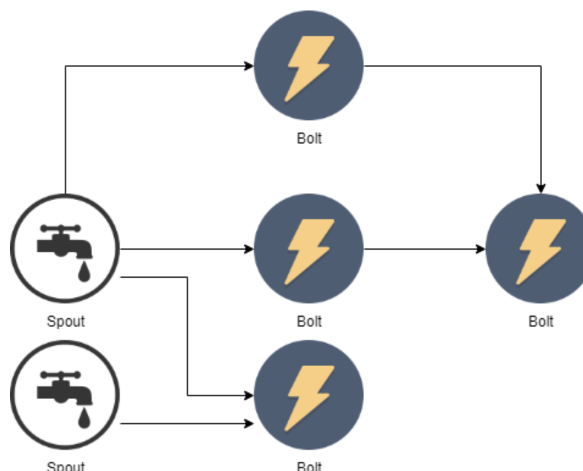

Figure 4.6: General example of a Storm topology.

In general, it is considered best practice to give each bolt a single responsibility. In the case where more complex processing is required, you would use a multitude of bolts to perform each of the individual required actions.

The big strength in Storm's processing capabilities lies within the fact that it is possible to assign multiple workers to perform certain tasks within the topology, creating a distributed processing paradigm. This will effectively decrease the time necessary to ingest incoming data.

Applications within the project Storm could be used in conjunction with Kafka to load data into the system. For every Kafka topic, we can create a Storm Spout that ingests data from Kafka into the Storm topology. From this point onward, we are able to lead the data through the correct steps in order to be fully ingested and stored in an HBase table.

The main advantage of using Storm is that you have free choice of the way in which you lead data through the topology. Each spout and bolt can have multiple exits, so you can decide to send data into a specific processing path before it is stored, allowing specific relevant operations to take place on the data before it ends up in HBase.

4.5.2 Apache Kafka

Apache Kafka is a software project that consists of several different layers, each layer having a separate responsibility. It is capable of acting as a message queue, processing streams of messages and storing messages in a distributed fashion.

Firstly, let's consider its storage and message queue capabilities. Apache Kafka employs the publish/subscribe (or pub/sub) paradigm, which allows software to pub-

lish messages to a so-called ”topic”. Applications that subscribe to this topic will then automatically receive the message that was sent, so that it can be processed. This subscriber can be an external application like Apache Storm, or it can be Kafka itself, as Kafka is also capable of processing data.

Incoming data is stored in a distributed fashion on a node in the cluster Kafka is running on. A node within this cluster in the context of this project is known as a Kafka broker.

Kafka also contains a library known as Kafka Streaming, which is its own stream processing library. Like Apache Storm, it is capable of distributedly processing incoming data. It may utilize external file storage mechanisms and database through Kafka Connect.

Kafka Connect is a framework in Kafka that lets it integrate with other systems. Kafka Connectors may be used to pull data from a data source, or push data towards a destination. The former is called a Source Connector, whereas the latter is known as a Sink Connector. There are many connectors available for different data sources and destinations. Some of these connectors are created and maintained by Kafka’s creator Confluent, whereas others are managed and maintained by the development community.

Applications within the project Kafka can be used as a method to enqueue streaming data before it enters the ETL process proper. Other processing frameworks with a connector to Kafka may then pull data from the queue in order to perform ETL operations on it.

4.5.3 Apache Flume

Apache Flume is a framework that specializes in the collection, aggregation and transferral of log data. It is capable of doing so for streaming data flows. An overview of the standard Flume data flow mechanism can be found in Figure 4.7.

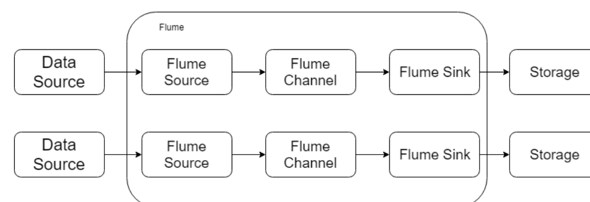

Figure 4.7: An overview of the standard Flume data flow mechanism

Flume is capable of consuming events that come from external sources. This source can be anything, from a web server to a standalone application. The source transfers the event data in a format that is understood by Flume, after which the data will flow through to a channel: A temporary passive data store where the data will stay until it is consumed by a Flume sink. The sink will then place the data into a persistent data store such as HDFS, HBase or Hive.

It is also possible for the sink to become the source for a new data flow, allowing Flume to aggregate multiple data streams together into a single data stream. Figure 4.8 shows an example of such an aggregated data flow mechanism.

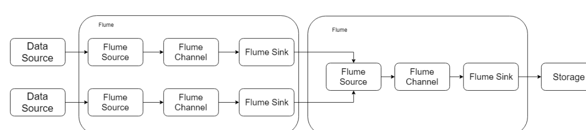

Figure 4.8: An overview of the Flume aggregated data flow mechanism

Applications within the project Flume would work very well in creating a streaming ETL pipeline in conjunction with Kafka. Flume has intrinsic agents which are able to pull data from Kafka topics and propagate them into the transformational parts of ETL in Flume.

4.5.4 Apache Hive

Apache Hive is a software project built on top of Hadoop for data warehousing purposes. It is capable of performing many different operations related to Hadoop, such as ETL/ELT, query execution and an SQL abstraction layer.

Hive has its own query language based on SQL called HiveQL. This allows users to write queries in SQL and have them come out and be executed as a Tez, Spark or MapReduce job. These queries are very powerful, as it allows Hive to be used as an ETL, query and/or data analysis tool, while preserving the advantages of distributed processing by using one of the three underlying engines.

Hive provides a few different storage formats, most notably including plain text files on HDFS, and HBase tables. Metadata can be stored in a small embedded RDBMS derived from Apache Derby, which used to be known as IBM CloudScape. This small embedded database can be queried with SQL, and by extension, with HiveQL. It's a handy tool for storing small bits of information that might help structure big data queries.

4.5.5 Apache Spark

Apache Spark is a popular cluster-computing framework used for several purposes. It contains a library for performing SQL queries called Spark SQL, a library for machine learning called Spark MLlib, a library for streaming applications called Spark Streaming and a library for creating graphical representations of data collections called Spark GraphX. All of these libraries are able to run on top of the central Spark library, which is named Spark Core.

Spark Core

Spark Core is the main library of Apache Spark, which means that every running application based on Spark will consist of a few core components. A Spark application will always contain a user-defined main function that is run when the program in executed. This application can then perform a set of parallel operations on a cluster. Spark also consists of a few main abstractions that it uses to represent information relevant to the application that is being run.

The first main abstraction is the Resilient Distributed Dataset (or RDD, for short), which is a dataset containing elements that can be processed in parallel. They are based on file in the Hadoop FileSystem (HDFS), but there are abstractions available for other storage systems. Most importantly for this project, there is one available for Apache HBase.

Spark Datasets

The Spark Datasets API is a library related to Apache Spark which functions as a bridge between Spark's RDD advantages such as strong typing, and Spark SQL's execution engine. Datasets can be created based on Spark-related Java objects, and then manipulated by using Java-based transformations, such as map and filter operations.

Spark DataFrame

The Spark DataFrame API is a slightly optimized representative collection of data organized into named columns. It is strikingly similar to a traditional relational table, and can be constructed from a variety of Hadoop-related objects, such as Hive tables, structured data files (HDFS) or Spark RDDs.

Applications within the project Spark could be used to ingest both batch and streaming data into the system. For batch ingestion, we can just use Spark Core, as it already has a library for linking to HBase, and we can just add the data as an input file to the Spark job.

For streaming ingestion, the Spark Streaming library can be used. As with previous streaming frameworks, it is possible to use Kafka as a queueing mechanism to retain streaming data until it is ready to be processed. Spark Streaming would then read this data from a Kafka topic and process it through the rest of the ETL mechanism.

4.5.6 Apache Sqoop

Apache Sqoop is a framework that facilitates the transferring of data from traditional Relational Database Management Systems (RDBMS) like MySQL databases, Microsoft SQL Server databases, et cetera, to a Hadoop cluster.

Sqoop consists of two separate sub-projects: Sqoop and Sqoop2. Sqoop2 was originally intended to be a successor to Sqoop, but it seems that development has on Sqoop2 has been rather slow, and a feature complete version is not available as of yet.

Sqoop has connectors for all major RDBMS, and supports the transferring of data from RDBMS to Hive and HBase. It does not support the reverse, however: It is not possible to transfer data directly from Hive or HBase to a RDBMS. For this, you would need to first export data from Hive or HBase to HDFS. Then you would be able to use Sqoop to transfer this HDFS data to a RDBMS.

On top of these features, Sqoop provides Kerberos security integration. This can be used to require users to authenticate themselves in order to run Sqoop jobs.

Applications within the project Sqoop may be used in order to move bulk or batch data from RDBMS to our system. It could come in handy whenever there is log data stored in a MySQL or Microsoft SQL database, but at the moment there is no such data, so we will leave Sqoop out of the consideration for the time being.

4.6 Querying

In this section, we will list the available technologies that are capable of querying data from HBase. We will list the strengths and weaknesses of each individual technology. By comparing these aspects and by checking how well each of them satisfies the requirements, we can select one or more technology and implement them into the solution.

In terms of distributed processing, there are quite a few techniques that Hadoop offers that facilitate querying data. Some of them are focused on providing an SQL-like interface for Hadoop-related software in order to make it easier to work with, whereas some others define their own domain-specific languages (DSLs) to handle this. In this section, we will highlight and describe some of the distributed processing frameworks that provide support for the Hadoop ecosystem.

First, let's go over the frameworks that provide SQL-like functionalities. Apache Hive is a widely adopted and active Apache project that supports the analysis of large data sets stored on HDFS and compatible systems. Users are presented with the possibility of writing SQL-like queries in a native query language called HiveQL. [26] These queries are then translated into a job that can run on one of three execution engines of Hadoop: MapReduce, Tez or Spark. [22, 38, 39]

Another example of such an SQL framework is Apache Impala (formerly known as Cloudera Impala). It is described as a technology that allows users to issue low-latency SQL queries to distributed storage systems as Apache HBase and HDFS. Since being integrated in the Hadoop toolkit, it has been modified to support the same file and data notations used by other software tools within Hadoop. [40]

Finally, Apache Phoenix is a framework that provides support for Hadoop while using HBase as a storage engine. It distinguishes itself from other technologies in this category by providing an SQL interface for the user, allowing them to perform table operations. These queries are compiled to native noSQL APIs rather than MapReduce jobs. It is claimed that this allows the user to build low-latency applications on noSQL stores. [41]

The sections below will provide more in-depth descriptions of some of the processing frameworks for Apache Hadoop, and determine how they may be integrated into the final solution by looking at which roles they may fulfill.

4.6.1 Apache Spark

Apache Spark is a library built for big data processing. It has modules for streaming data and SQL. It is capable of processing work over a hundred times faster than traditional Hadoop/MapReduce paradigms, and does so by using several paradigms for scheduling, query optimization and program execution.

Firstly, Spark performs its scheduling by using Directed Acyclic Graph (DAG) technology. It uses stage-oriented scheduling, which means that every step that needs to be executed is partitioned and allocated to a so-called task set. These task sets are logically distributed between the workers, such that it requires as little data transfer as possible during execution. [42]

Spark has recently shipped a new version of its software with a query optimization framework called Catalyst. This framework leverages statistics collected about each individual pieces of data, which are stored in order to efficiently execute queries. These statistics exist on the table-level, such as cardinality of relations between tables, as well as on the column-level, such as numbers of distinct values, maximum and minimum values and average and maximum field length. [43]

There are three different APIs available within Spark that allow for data processing: The Spark RDD API, the Spark Dataset API and the Spark DataFrame API. Of these three, only the RDD API existed in Spark, whereas the other two were introduced in Spark 2. The latter two are expected to be more optimized in terms of dealing with data. This will be further researched in the performance comparison experiment in Chapter 6.

4.6.1.1 Spark DataFrame Example

The following listing shows a Scala example of how a DataFrame can be created in Apache Spark code, in combination with the use of Apache HBase as a storage mechanism. The first course of action is to define a catalog, which determines the table and accompanying columns that Spark will query for in HBase.

```
def catalog = s"""{
|"table":{"namespace":"default",␣"name":"table1"},
|"rowkey":"key",
|"columns":{
|"col0":{"cf":"rowkey",␣"col":"key",␣"type":"string"},
|"col1":{"cf":"cf1",␣"col":"col1",␣"type":"boolean"},
|"col2":{"cf":"cf2",␣"col":"col2",␣"type":"double"},
|"col3":{"cf":"cf3",␣"col":"col3",␣"type":"float"},
|"col4":{"cf":"cf4",␣"col":"col4",␣"type":"int"},
|"col5":{"cf":"cf5",␣"col":"col5",␣"type":"bigint"},
|"col6":{"cf":"cf6",␣"col":"col6",␣"type":"smallint"},
|"col7":{"cf":"cf7",␣"col":"col7",␣"type":"string"},
|"col8":{"cf":"cf8",␣"col":"col8",␣"type":"tinyint"}
|}
|}""".stripMargin
```

Listing 4.3: Sample Spark DataFrame HBase table catalog [44]

Of course, such a catalog can also be created automatically from a pre-defined specification. A DataFrame can be created from this catalog, which can then be queried. It's possible to use an SQL query directly on a DataFrame:

```
val df = withCatalog(catalog)
df.createOrReplaceTempView("table")
sqlContext.sql("select␣count(col1)␣from␣table").show
```

Listing 4.4: Sample Spark DataFrame HBase query [44]

4.6.1.2 Spark Dataset Example

The Spark DataSet API is quite similar to the DataFrame API. The main difference is that instead of defining a catalog like you would for DataFrames, you define a JVM object with the representative fields in them. Then, you can have Spark read data and cast that data to a DataSet containing those JVM objects, which then in turn contain the data from HBase. An example of this is given below.

```
case class DeviceIoTData (
battery\_level: Long,
c02\_level: Long,
cca2: String,
cca3: String,
cn: String,
device\_id: Long,
device\_name: String,
humidity: Long,
ip: String,
latitude: Double,
longitude: Double,
scale: String,
temp: Long,
timestamp: Long
)

// read the json file and create the dataset from the
// case class DeviceIoTData
// ds is now a collection of JVM Scala objects DeviceIoTData
val ds = spark.read.json("/databricks-public-datasets/data/iot/
    ↪ iot_devices.json").as[DeviceIoTData]
```

Listing 4.5: Spark Dataset: Example of loading data into JVM object [45]

As can be seen, Spark reads the JSON input file and casts the data to a DataSet of DeviceIoTData objects. Again, the goal is to convert this prototype to Java in the remaining month.

4.6.2 Apache Hive

Apache Hive is a software project that offers query and analysis capabilities on top of Hadoop. It provides the means of managing large datasets that exist in distributed storage systems such as HDFS and HBase by using SQL. Users can run queries through a command line tool, or they can use a JDBC driver to use Hive. [26] Apache Hive is comparable to Apache Pig, a scripting language based on dataflows, except for the fact that Hive caters more to users that are more accustomed to using database queries as an interface.

4.6.3 Apache Phoenix

Much like Apache Hive, Apache Phoenix is a software project that provides SQL-like functionalities on top of Apache Hadoop. It uses HBase as a backing store, which means it can translate the SQL queries that a lot of developers are used to, and ports them to a NoSQL database. On top of this, Phoenix is integrated with many other software projects from the Hadoop ecosystem, including Spark, Hive and MapReduce. [41]

How Phoenix works internally is as follows: A user provides Phoenix with a query in SQL. This query is compiled by Phoenix and translated into a set of HBase Scan operations. The result is returned as a regular JDBC output. One of the great strengths of Phoenix working like this is the fact that Phoenix does not need to know anything about your schema to run SQL queries.

Chapter 5

System implementation

In the previous chapter, we compared technologies in various fields to decide on a set of technologies to use in the final solution. This chapter describes in detail how each individual technology was implemented. These descriptions will range from integration steps to implementation code examples. I will also mention any notable problems or conflicts that I ran into along the way and describe how I solved them.

Finally, this chapter will also describe examples of how the software may be utilized by users to store data on the system, as well as how they may retrieve data. Each of these cases will be explained with the help of several examples, each of which deals with the functional requirements of the specific part described in the previous chapter.

5.1 Storage Mechanism

In this section, we will describe how we implemented storage mechanisms in the final solution. We will briefly reiterate the choices that have been made, and describe in detail how the chosen technologies are integrated into the overall design. Next, we will describe the steps that have been taken to install and configure the storage system. We will also describe the data models that have been set in place and the rationale behind them. Each section will also contain a reference to some appendices that describe how to set the system up. This was done so that Thales employees can easily replicate the installation without having to re-invent the wheel.

The first subsection will describe how HBase has been integrated into the system, and the data model we use for it. The second subsection will deal with the auxiliary and support data, which is being stored in MySQL, a smaller-scale relational database management system (RDBMS).

5.1.1 HBase

This section will deal with the way HBase was integrated into the final solution. As was described in the chapter about the final design, HBase is a distributed key-value store that runs on top of HDFS. It is integrated into the toolkit surrounding Apache Hadoop, and has support from most, if not all related technologies.

As has been declared before, HBase is based on Google's BigTable. The idea behind this concept is that HBase is able to store large amounts of data in a sparse format. This means that space is not allocated for each data column as a whole, but rather for each field. This is also the reason why people prefer to refer to HBase as a key-value store, rather than a database.

In our system, HBase will be used to store all radar data. First, let's review some of the characteristics of this data. From the requirements, we know that it should be possible to store radar data at different levels of granularity. Some examples of this are:

- Raw radar data, which can consist of raw video.
- Hits, which are returning signals that are above the detection threshold.
- Plots, which are positions of confirmed entities.
- Tracks, which is the trajectory of a confirmed entity over time.

All of these levels of granularity are messages that are output by a radar. These messages must be ingested into the system in order to create a complete data store of all recorded radar data.

The fields and structure of radar log messages is known beforehand. Therefore, it is possible to create a specification of the messages and validate any incoming data. For this project, such a specification is considered auxiliary information, and therefore it will be described in the Section 5.1.2, where the structure of auxiliary data is discussed.

Let us take a look at how we structure data in HBase. Since the data is all closely related to each other, we have decided to use a single table in order to store data from all radar data types. This comes with a few advantages, but also requires some careful design in order to make sure that it is still efficient.

One of the advantages is that you never have to distinguish between tables. You can always query the same table, because all of the data is in there.

Another advantage is the fact that you can ensure data proximity, which means that data which is related to each other, can be stored close to each other in the cluster. The data will reside in the same HBase Region. Since HBase depends on

the design of the row key in order to place it into a Region, we need to carefully think about the design of the row key.

In general, it is inadvisable to use monotonically increasing row keys (such as timestamps). Since measurements may contain data from multiple radars at once, we should not use timestamps as leading factor in a row key. After all, we would like to keep data from different radars separate, while keeping data from the same radar in close proximity to each other. If we do this, we will able to query this data more efficiently, since we are more likely to query a lot of data from the same table.

What we can do, however, is to use the timestamp as a deciding factor at the end of the row key, to distinguish between messages. Since we want to split data proximity by radar type, we will use characteristics from the measurement location and the sensor itself to lead the row key.

Figure 5.1 shows the design of this table in HBase.

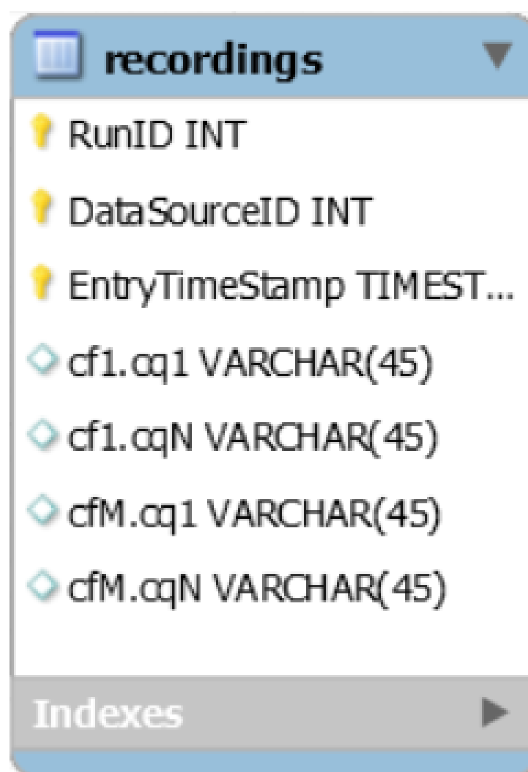

Figure 5.1: HBase Table Design

First, let's take a look at the design of the row key. As mentioned before, we did not want the timestamp to be the deciding factor in the localization of the data. Rather, we would like to format based on actual measurement data related to the place of measurement, as well as the radar that was used.

In order to explain the structure of the row key, we will need to explain some of the terminology used here:

- **Run:** A run is an event at which the measurements took place. This could be a mission identifier when used in practice, or a measurement campaign identifier when used for testing purposes. These identifiers are decided upon by Thales.
- **Data Source:** The data source is the type of radar used for the measurement in the stored row. This information will come from the messages sent by the radar to the cluster.
- **Entry Time Stamp:** The entry time stamp is the uniform timestamp that is allocated after the message has been sent to the system, after it has been processed by the ETL tools and is ready to be stored.

By using these three in sequence as the row key, we can ensure a few things that are important. Firstly, we can ensure that each of the row keys is unique. This is critical, because writing data to columns in the same row key may result in the overwriting of data, which we do not want for obvious reasons. Secondly, we have ensured that data related to each other, both from the same run and the same radar, will be stored within regions in close proximity to each other, if not the same region.

The way this design works for HBase's column structure is that column families will represent the sensors that generate the data, and column qualifiers will contain the data entries. For example, if we have two radar types called Radar1 and Radar2, the fields list in the big HBase table may consist of:

- Radar1
  - Radar1:field1
  - Radar1:field2
  - Radar1:field3
  - ...
- Radar2
  - Radar2:field1
  - Radar2:field2
  - Radar2:field3
  - ...

5.1.2 MySQL

In order to store auxiliary and support information, such as user accounts, permissions and metadata related to querying HBase, we will be using a small-scale MySQL database. This section will describe how we store information in this MySQL database, what the characteristics of this data are and how they will be used.

First, let's look at the responsibilities this MySQL database will have when it comes to storing data in HBase, and querying data from the data set:

- The first responsibility should be the capability of storing and holding a data specification for each individual radar.
- Secondly, it should be possible to provide a base specification, which will be used for the conversion of data between differing radar data types.
- Thirdly, it should be possible to provide relations between columns of different radar data types, and provide a way to convert all of them to a field from the base data type.

Therefore, it should be possible for users to provide this data specification through an interface, which will then be persisted in the MySQL database. As soon as the ETL receives some data, it will retrieve the related specification and check whether all of the data entries match.

From this can be deduced that there needs to be a possibility to store a set of column family names, which represent sensor types. These are stored in the *column_families* table. Each of these sensor types have their own data specifications. Each of these specifications can be represented by a stream of messages with the same structure. The *streams* table represents this. Each entry in this specification will be represented by a column qualifier, contained in the *column_qualifiers* table. In order to store the full specification in MySQL, we will need to add a table containing column qualifiers, each of which is linked to a column family. You may see an example of this structure in Figure 5.2.

To support the adaptability portion of the system, there also needs to be a way to add relations to this MySQL database. The table representing column qualifiers can be used to link fields to a common base specification, alongside a conversion formula that converts each of the relevant fields to a common unit or format.

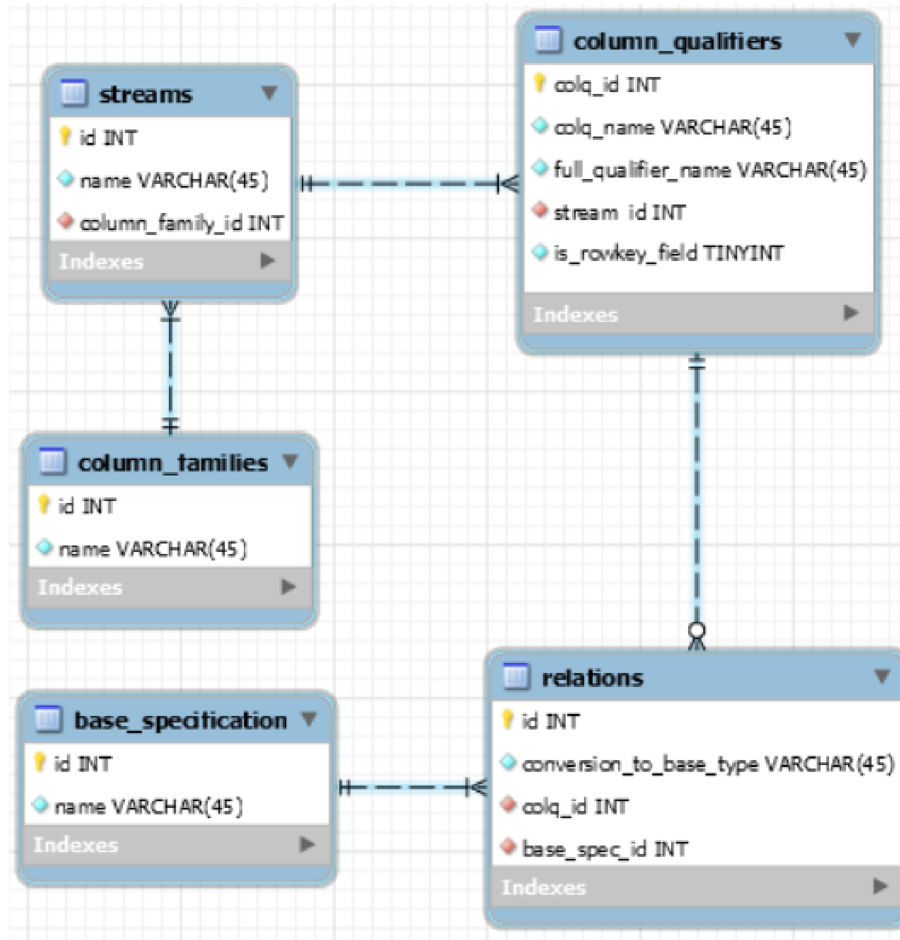

Figure 5.2: MySQL Table Design

5.2 ETL

This section will describe the implementation of the ETL tool(s) used for this project. Several prototypes were created, each of which uses a different combination of technologies from the Apache Hadoop software ecosystem.

5.2.1 Prototype 1: Apache Kafka + Apache Storm

The first prototype for ETL was made with the technologies Apache Kafka and Apache Storm. An overview of these technologies used on top of the envisioned architecture from Section 3.2.3 can be found in Figure 5.3. As shown, Apache Kafka is used to queue streaming data, which is then ingested into an Apache Storm topology.

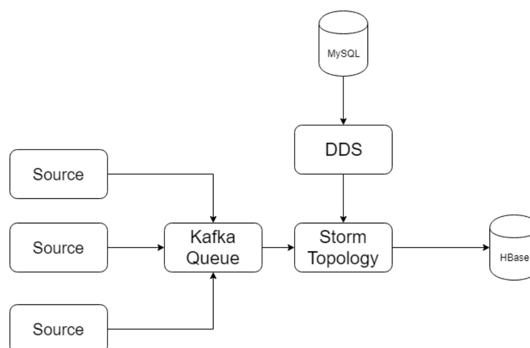

Figure 5.3: ETL Implementation Architecture Overview

As mentioned in Section 4.5.2, Kafka is a technology that can be used for a variety of applications. In this prototype, we will use it as a distributed publisher/-subscriber queue for incoming radar data. Radars can publish their log data into the queue, after which it will exist there until it is processed.

In order to publish data into Kafka, it needs to be sent to a specific topic. This topic is defined by an identifier. In the case of this prototype, each radar topic is identified by the same name that is used to define the column family in HBase.

Once data has been published into Kafka, it needs to be picked up by a Spout in the Storm topology. This Spout is a part of the topology that specifically subscribes to a Kafka topic, and passes data onward to the rest of the topology.

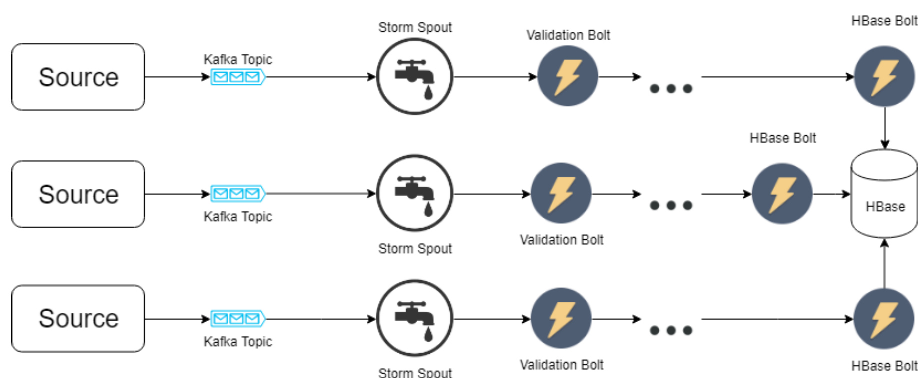

Figure 5.4: Overview of prototype implementation of Kafka and Storm for ETL

The rest of the topology will consist of bolts that use the data to perform some action. For example, in order to keep data clean and correct, validation will need to be applied to incoming data. Since each data stream in the topology is responsible for its own radar data type, a Bolt reponsible for validation can retrieve the specification from the MySQL database and use it to check the incoming data.

Should this data be deemed valid, then it can be passed along to the next node in the topology. By giving each individual Bolt a certain responsibility, a modular ETL system is created, where Bolts can be added and removed at any given point

in time. At the end of the topology, there is always an HBase Bolt, which persists resulting data in the big data storage system. An overview of this architecture can be found in Figure 5.4.

The main part of the topology is the custom MultiStreamIngestionTopology class. It is the class that is responsible for creating spouts and bolts and connecting them together to form the overall topology. Multiple Kafka topics can be passed as arguments to the execution of this class, and each of them will create a Spout in the topology.

First, a loop is created that goes through each of the topics passed to the class. The following actions take place for each loop. Initially, the MySQL database is queried for the topic name, to retrieve the column family ID in the MySQL database. This is necessary so that the streams (a.k.a. the message formats) for that particular radar can be retrieved. Remember that the column family name represents a specific radar in this setup.

```
int columnFamilyId = MySQLHelper.getColumnFamilyIdByName(arg);
List<Stream> streams = MySQLHelper.getColumnFamilyStreams(
    ↪ columnFamilyId);
```

Listing 5.1: MySQL queries for loading message streams

Next, a Storm bolt is created. For this purpose, I created a class called MultiStreamStormBolt that implements IRichBolt, so that I could easily accommodate for multiple streams of messages passing through the bolt.

```
MultiStreamStormBolt bolt = new MultiStreamStormBolt(streams);
```

Listing 5.2: Creating an instance of the Custom MultiStreamStormBolt class

After creating this bolt, mappers need to be created in order to map messages to the HBase table. Since streams already contain the fields that need to be present in each message, I can simply create mappers based on the information that is present. The MultiRowkeyColumnHBaseMapper is also a custom implementation of the HBaseMapper interface, so that I can use multiple columns to create a combined row key.

```
Map<String, HBaseMapper> mappers = new HashMap<>();
for (Stream stream : streams) {
        List<String> rowKeyFields = stream.getRowKeyFields();

        MultiRowkeyColumnHBaseMapper mapper = new
           ↪ MultiRowkeyColumnHBaseMapper()
        .withRowKeyFields(new Fields(rowKeyFields))
        .withColumnFields(new Fields(stream.getFields()))
        .withColumnFamily(arg);

        mappers.put(stream.getName(), mapper);
}
```

Listing 5.3: Creation of custom HBase mappers

Next, the HBase bolt needs to be created. The information necessary is the table name and the set of mappers we have just created. I needed to create a custom implementation of the HBaseBolt called CustomHBaseBolt in order to accommodate for the multiple mappers, a functionality that the regular HBaseBolt does not have out of the box.

```
CustomHBaseBolt hbase = new CustomHBaseBolt(tableName, mappers)
.withConfigKey();
```

Listing 5.4: Creation of custom HBaseBolt based on mappers

Finally, the spouts and bolts need to be hooked up so that the messages are passed down the topology. I use an incrementing variable *i* to separate this particular topology route from others, since we are creating new Spouts and Bolts for each topic/column family.

```
builder.setSpout(JSON_SPOUT + i, new KafkaSpout<>(buildSpoutConfig(arg
   ↪ )), 1);
builder.setBolt(STORM_BOLT + i, bolt, 1).shuffleGrouping(JSON_SPOUT +
   ↪ i);
BoltDeclarer declarer = builder.setBolt(HBASE_BOLT + i, hbase, 1);

for (String streamName : streams.stream().map(Stream::getName).collect
   ↪ (Collectors.toList())) {
        declarer.shuffleGrouping(STORM_BOLT + i, streamName);
}
```

Listing 5.5: Creation of topology order and grouping

Finally, I will explain some of the changes I made to the default Storm Bolt. In order to accommodate multiple streams, each message type that belongs to a topic is stored in that topic's MultiStreamStormBolt instance. When a tuple passes into the Bolt, it checks each of the stream for a matching specification.

```
for(Stream s : streams){
        boolean match = true;
        for (Object o : json.keySet()) {
                if (!s.getFields().contains(o.toString())) {
                        match = false;
                }
        }

        if(match){
                streamToEmit = s;
                break;
        }
}
```

Listing 5.6: Checking if incoming message corresponds to any expected message format

If no stream is found that matches, the tuple will automatically fail, as shown below:

```
if (streamToEmit == null) {
        LOG.info("FAIL␣-␣No␣stream␣matches:␣" + tuple);
        collector.fail(tuple);
}
```

Listing 5.7: Failing the tuple in case no matching message format is found

If a matching stream is found, the fields are added to the collector and emitted to the HBase Bolt.

```
Values v = new Values();
for(String field : streamToEmit.getFields()){
v.add(json.get(field));
}

collector.emit(streamToEmit.getName(), tuple, v);
collector.ack(tuple);
```

Listing 5.8: Emitting the tuple to the next bolt after adding the values in the message to the emit data

5.2.2 Prototype 2: Apache Spark

The second prototype that has been implemented in order to perform ETL into HBase was created with Apache Spark. Spark offers a library called Spark Streaming, which allows it to process data that is passed to it via streams. In the demonstrator, this has been done by making Spark subscribe to a Kafka queue. In order to call the Spark Streaming ETL prototype class, a Kafka topic needs to be passed as a parameter in order to make it subscribe to it.

The first thing that needs to be done is to initialize the Spark Context, and the Spark Streaming context:

```
SparkConf sparkConf = new SparkConf().setAppName("Spark␣Streaming␣ETL␣
    ↪ Test");

JavaSparkContext jsc = new JavaSparkContext(sparkConf);

JavaStreamingContext jssc = new JavaStreamingContext(jsc, new Duration
    ↪ (1000));
```

Listing 5.9: Creation of Spark Contexts for Streaming ETL

Next, Spark is told to subscribe to the passed Kafka topics, and the tuple records that pass through the Kafka queue are mapped to a singular stream of JSON messages.

```
JavaInputDStream<ConsumerRecord<String, String>> directKafkaStream =
KafkaUtils.createDirectStream(jssc, LocationStrategies.
    ↪ PreferConsistent(),
ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));

JavaDStream<String> messages = directKafkaStream.map((ConsumerRecord<
    ↪ String, String> record) -> record.value());
```

Listing 5.10: Creation of input stream based on Kafka topic in Spark Streaming

Mapping this stream to HBase is very simple. Simply create the HBase context based on the existing Spark context, and perform streaming bulk puts based on the messages that are received. The described PutFunction creates an HBase Put object containing the record.

```
JavaHBaseContext hBaseContext = new JavaHBaseContext(jsc, conf);

hBaseContext.streamBulkPut(messages, TableName.valueOf(tableName), new
    ↪  PutFunction(topicName));
```

Listing 5.11: Creation of HBaseContext and setting up Streaming bulk Puts to HBase in Spark Streaming

Finally, the Streaming context is started, and is told to await termination. It is possible to pass a parameter in order to tell the prototype to run for a certain amount of milliseconds, but it is omitted in this example.

```
jssc.start();
jssc.awaitTermination();
```

Listing 5.12: Initialization of Spark Streaming Context in Java

5.3 Querying

This section will describe how queries have been implemented into the system. After the research into existing technologies in querying in Section 4.6, the decision has been made to use Apache Spark as our primary querying engine. The main reason for this are the fact that Spark is widely considered to be the most versatile and accepted distributed processing technology. Since we cannot explore every single technology in the timespan of this project, Spark was the most suitable choice to try.

There are three APIs that Spark can utilize to query a storage system: The RDD, DataFrame and DataSet API. This section will show how each of these three prototypes have been implemented, how they function and how well they perform.

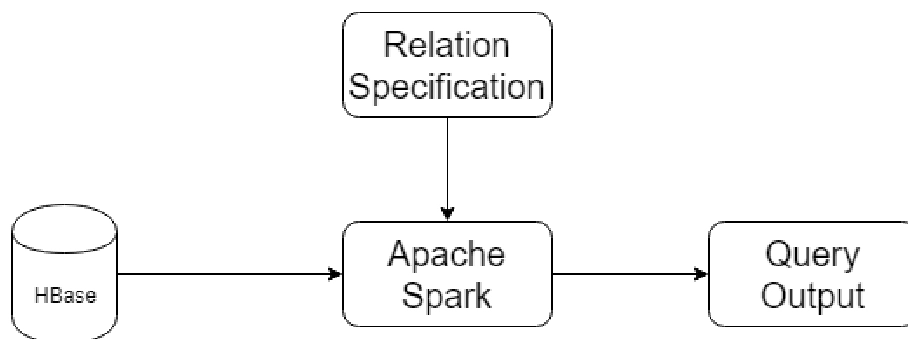

Figure 5.5: Querying Implementation Architecture Overview

5.3.1 Prototype 1: Spark RDD Prototype

The first prototype queries HBase with the help of the Spark RDD API. A Resilient Distributed Dataset (RDD) can be created by referencing external data sources, such as HDFS and Apache HBase. Such a dataset is a representation of the data that is present in the referenced datastore.

This representation can then be queried by applying transformations or actions to it. Such a transformation could be a filter, a union with another dataset, or optimization of dataset properties. At the end, the data that satisfies any filters will be

returned as the result set. A graphical description of this process can be found in Figure 5.6.

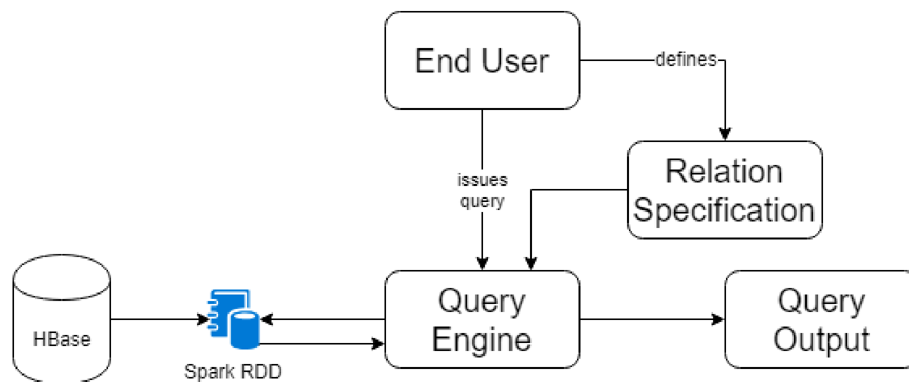

Figure 5.6: Spark RDD Implementation Overview

In the case of this implementation, users can issue queries to retrieve data from column qualifiers they specify. Filters may be applied to further suit the relevance of the queried data. Spark will then retrieve the data based on the given query. A code sample of the SparkQuery class can be found in Appendix **??**.

In addition to retrieving this data from the RDD, Spark will apply any necessary conversions to the data according to the relation specification defined for the queried data sources. These relation specifications can be defined in the user interface, as will be shown in Section 5.4.

In the SparkQuery class, whenever a query is issued by a user, it first creates a Context, or a connection to HBase. An RDD is then created, and is then filtered through the qualifiers that need to be queried. See Listing 5.13 for an example.

```
JavaHBaseContext hBaseContext = new JavaHBaseContext(jsc, conf
    ↪ );

JavaRDD<Tuple2<ImmutableBytesWritable, Result>> rdd =
    ↪ hBaseContext.hbaseRDD(TableName.valueOf(tableName), scan
    ↪ );

List<String> results = rdd.map(new ScanConvertFunction(
    ↪ qualifiers)).collect();
```

Listing 5.13: Spark RDD Scan on HBase

The ScanConvertFunction then checks for each returned column if it is contained in the query. If it is, it will be added to the query result, and will also apply conversions from the relation specification if necessary.

```
if(result.containsColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(
    ↪ columnQualifier))){
        byte[] field = result.getValue(Bytes.toBytes(columnFamily),
            ↪ Bytes.toBytes(columnQualifier));

        field = Converter.convert(qualifier, field);

        innerJson.put(columnQualifier, Bytes.toString(field));
}
```

Listing 5.14: Check if column is required. If it is, apply any conversions (if necessary) and put it in the return JSON object

5.3.2 Prototype 2: Spark DataFrame Prototype

The Hadoop distribution that is being used (Hortonworks HDP 2.6.1.0) supports its own version of the Spark-HBase connector, located at https://github.com/hortonworks-spark/shc. With the DataFrame API, it becomes possible to define a catalog that describes the table structure. The Java example below shows how column data pulled from the data specification in MySQL can be inserted into the catalog. The tableName and queryString variables that are being passed to the constructor of the GenericDataFrameQuery class are both user-defined input.

```
GenericDataFrameQuery query = new GenericDataFrameQuery(tableName,
    ↪ columns, queryString);
query.execute();
```

Listing 5.15: Creation of a custom DataFrame query object, and calling its execute() method

The GenericDataFrameQuery class is also a custom class that handles the execution of the sent query and then stores the result on HDFS. The constructor creates the HBase catalog through the also custom HBaseTableCatalogBuilder class.

```
public GenericDataFrameQuery(String tableName, Map<String, String>
    ↪ columns, String query){
        builder = new HBaseTableCatalogBuilder(tableName);
        builder.addColumns(columns);

        this.query = query;
}
```

Listing 5.16: Constructor of the custom DataFrame query class. Creates a Catalog builder for HBase and adds the passed columns to it

The execute() method of this class then creates the SparkSession and executes the query through the use of a DataFrame:

```
public void execute(){
        SparkSession spark = SparkSession
                                                .builder()
                                                .appName(query)
                                                .getOrCreate();

        Map<String, String> catalog = builder.getCatalog();

        Dataset<Row> df = spark
                                        .sqlContext()
                                        .read()
                                        .options(catalog)
                                        .format("org.apache.spark.sql.
                                            ↪ execution.datasources.
                                            ↪ hbase")
                                        .load();

        df.createOrReplaceTempView(builder.getTableName());
        df = spark.sqlContext().sql(query);

        df.write()
                .mode(SaveMode.Append)
                .format("json")
                .save("/home/alex/output/dataframe");
}
```

Listing 5.17: execute() method of the custom DataFrame query class

The HBaseTableCatalogBuilder contains a nested JSONObject that describes the overall structure of the table. The top-level object contains a sub-object describ-

ing the table, and a sub-object describing the columns that need to be queried in that table. The addColumns method can be used to add a collection of columns to the column object:

```
public void addColumns(Map<String, String> columns){
        columns.forEach((k,v) -> {
                JSONObject col = new JSONObject();

                String columnFamily = k.split(":")[0];
                String columnName = k.split(":")[1];

                col.put("cf", columnFamily);
                col.put("col", columnName);
                col.put("type", v);
                colObj.put(columnName, col);
        });
}
```

Listing 5.18: addColumns() method of the custom HBaseTableCatalogBuilder class

The ”cf”, ”col” and ”type” keys are internal conventions that are used to describe the catalog for HBase to understand. When the catalog is retrieved through the getCatalog() method, the top-level JSONObject (along with its sub-objects) is converted into a Map containing the HBase table catalog notation:

```
public Map<String, String> getCatalog(){
        mainObj.put("columns", colObj);

        Map<String,String> catalog = new HashMap<>();
        catalog.put(HBaseTableCatalog.tableCatalog(), mainObj.toString
           ↪ ());

        return catalog;
}
```

Listing 5.19: getCatalog() method of the custom HBaseTableCatalogBuilder class

5.3.3 Prototype 3: Spark DataSet Prototype

The Spark DataSet API is quite similar to the DataFrame API. The main difference is that instead of defining a catalog like you would for DataFrames, you define a JVM object with the representative fields in them. Then, you can have Spark read data and cast that data to a DataSet containing those JVM objects, which then in turn contain the data from HBase.

In order to use the Dataset API to query HBase, a serializable class needs to be created which can hold the fields that will be queried. The GenericDataSetQuery is almost identical to the GenericDataFrameQuery class, except for the fact that the Dataset class contains a few extra notations in the execute() method. Note the following example for the serializable class NS106.

```
public void execute(){
        Encoder<NS106> ns106Encoder = Encoders.bean(NS106.class);
        SparkSession spark = SparkSession
                                        .builder()
                                        .appName(query)
                                        .getOrCreate();

        Map<String, String> catalog = builder.getCatalog();

        Dataset<NS106> ds = spark
                                        .sqlContext()
                                        .read()
                                        .options(catalog)
                                        .format("org.apache.
                                           ↪ spark.sql.
                                           ↪ execution.
                                           ↪ datasources.
                                           ↪ hbase")
                                        .load()
                                        .as(ns106Encoder);

        ds.createOrReplaceTempView(builder.getTableName());
        ds = spark.sqlContext().sql(query);

        ds.write()
                .mode(SaveMode.Append)
                .format("json")
                .save("/home/alex/output/dataset");
}
```

Listing 5.20: execute() method of the custom Dataset query class

Note how this execute() method is almost identical to the one used for the DataFrame API, with the only execeptions being the need for an Encoder class and the Dataset being typed with the serializable class instead of the previously used untyped Dataset <Row>.

5.4 Adaptability

It has been made clear that in order to cross-reference data from multiple different radar data types, there is a need for defining relations so that conversions between related, non-uniform fields can be applied. This section describes how relation specifications have been implemented, and how they can be defined by end users in order to gain uniform query output data.

First, the different specifications will be considered. The possibility to create a base specification has been implemented, in order to allow any other data type to convert to it. This base specification will need to be uniform, i.e. for each radar data type that differs from the base specification in a specific type of field, there is a defined relation with an attached conversion formula.

The choice has been made to create a custom implementation for this particular part of the system. This is because there are no solutions readily available to automatically create these conversions, and some degree of user-defined conversion will be necessary. In order to facilitate this, a web-based user interface has been created that will allow for the definition of the base specification, and any necessary conversions that need to be applied to data as it is queried.

The web-based UI has been set up with the help of PHP framework Laravel (https://www.laravel.com). The RESTful and MVC approaches that Laravel offers out of the box makes it simple and efficient to perform CRUD operations in order to create the required support data.

Laravel employs a system called Migrations, which allows developers to define table structures in a definition file. When running migrations, the table structure will automatically be added into an associated RDBMS. In our case this is MySQL, as mentioned in Section 5.1.

Creating the Content Management System that allows the manipulation of content, such as the creation of relations in the database, was done with the help of Laravel Voyager. Voyager is an admin panel integrated with a CMS that makes it easy to perform any CRUD operation, automatically adding web pages that facilitate this, and taking care of the bindings to RESTful controllers internally.

Relation conversions have also been integrated with the Spark prototypes. Each query to an entry in the HBase table will result in a call to the Converter class (See Appendix B), which checks if any of the fields in a returned row require conversion to the base specification, and if so, converts them.

5.5 Access Control

With the time left, some research went into the implementation of access control. Several things needed to be possible in the final solution, mainly the restriction of access to (parts of) the dataset for specified users, as well as the possibility of encrypting specific parts of the data.

In order to ensure the restriction of access to the dataset, HBase supports its own form of access control through the functions Secure RPC and Access Control. By modifying the hbase-site.xml file on each node like below, the AccessController coprocessor in HBase is activated. This makes it so all client access to HBase needs to be activated. Access to data can then be granted to users on a table or even column family basis.

```
<property>
        <name>hbase.coprocessor.master.classes</name>
        <value>org.apache.hadoop.hbase.security.access.
           ↪ AccessController</value>
</property>
<property>
        <name>hbase.coprocessor.region.classes</name>
        <value>org.apache.hadoop.hbase.security.token.TokenProvider,
        org.apache.hadoop.hbase.security.access.AccessController</
           ↪ value>
</property>
```

Listing 5.21: HBase configuration for security

As shown in the Access Control documentation for HBase, it is easy to grant user permissions through the shell extension for HBase. [46]

Finding a solution for the encryption problem proved to be a bit more difficult. Native encryption in HBase happens at the level of HFiles, which may contain overlap between subsets of the data, parts of which may need to be encrypted, and parts of which may not. Evidently, this does not entirely support the requirement, because using it would result in data being encrypted where this is not necessary or desirable.

A potential solution for this problem would be to handle encryption in a custom way, for example via the previously mentioned Ciphertext-Policy Attribute-Based Encryption (CP-ABE) technology. With this, encrypting individual cells or column families would be possible. The only thing that would need to happen additionally is to tie a CP-ABE policy to the Access Control domain within HBase, to ensure that users that attempt to retrieve the data need to match policies in order to decrypt the original data.

Chapter 6

Validation

This chapter will describe the validation of the implemented demonstrator. After the system has been developed in the form of several separate prototypes, there is a need to check whether or not these prototypes satisfy the requirements that have been set. Validation of the demonstrator with the help of the requirements is two-fold: Firstly, the validity of the implementation of functional requirements can be checked by performing requirements validation, i.e. simply checking whether or not the prototype is capable of performing specific functional actions. Secondly, the non-functional requirements can be checked by performing some experiments that compare the values of certain quality aspects.

6.1 Cluster setup

Before describing the validation of the software against the requirements, it is important to know what cluster architecture was used. For the purpose of this project, i.e. the development of a demonstrator, a minimal cluster architecture was used. Table 6.1 shows an overview of the hardware used in the sample cluster.

CPUs	3x Intel Xeon E5520 (Quad-core) @ 2.27GHz
Total RAM	26 GB DDR3
Ethernet Connection	200 Mbps
Total Cluster Storage	1 TB
Total YARN Memory	11 GB
NameNode Heap	1 GB
HDFS Heap	1 GB
ResourceManager Heap	1 GB
Master Nodes/NameNodes	1
Slave Nodes/DataNodes	2

Table 6.1: Cluster Hardware Description Table

The requirements validation, as well as the experiments in Section 6.3 are conducted on a data set of about 4 GB, containing sample data of both the NS106 and the Squire radar.

6.2 Requirements validation

This section will describe the validation of the overall requirements. Functional requirements are requirements that can be easily validated simply by checking whether the designed and implemented system is capable of performing the required functionalities. This section will show how these actions can be performed with the help of several examples including experiment input, expected output and actual output.

6.2.1 Storage

In this section, the requirements for the storage mechanism will be validated. In order to do this, the chosen solution has to be checked against the functional requirements. The chosen storage solution is Apache HBase, with a single table that is capable of storing all sensor data.

STO-FR1: The system must be capable of storing different types of sensor data.

As has been described, sensor data is stored in a single table in the final solution. Each individual sensor type is assigned a column family in HBase, and a specification may be defined to specify which fields are valid for a specific sensor type.

STO-FR2: The system must be able to store data of which the format is not known at the time of system design.

The chosen architecture uses Apache HBase as a backing storage. HBase is agnostic when it comes to the structure of the data it contains. It simply contains a set of tables, which consist of a set of column families, which in turn contain a set of column qualifiers. HBase does not need to know in advance which column families and qualifiers exist in a table, allowing it to be versatile, and allowing fields to be added to the table on-the-fly.

STO-FR3: The system should be schema-less.

For the same reason as the above requirement, HBase does not need to have a pre-defined schema in order to store data. HBase is capable of adding new column qualifiers on-the-fly and storing data in the related rows. Additionally, there is no unnecessary overhead for each column, as HBase tables are sparse. This is due to the fact that HBase is in fact a distributed key-value store, rather than having an actual table structure.

STO-FR4: It should be trivial to translate between the input data format (e.g. JSON, XML, binary log files) and the storage format used by the database.

In the ETL prototype, ways have been added to use well-known data formats in order to load data into HBase. JSON/XML are simple data structures that can simply be looped through in order to get the fields and related data, which can then be passed along into the ETL process. Thales has in-house tools which can be used to convert binary sensor log files to a JSON and/or XML format, allowing these to be easily parsed as well.

6.2.2 ETL

ETL-FR1: The system should be able to ingest streaming data.

As can be seen in Section 5.2, an ETL prototype has been developed with the help of Apache Kafka and Apache Storm. Data is first inserted into a distributed Kafka queue, after which a Storm Spout will receive it and propagate it through the rest of the topology.

ETL-FR2: The system should be able to ingest batch data.

As shown in Section 5.2, prototypes for batch ETL have been implemented in Apache Hive and Apache Spark. Both of these technologies provide ways to ingest data into Apache HBase. Their performance will be compared in Section 6.3.

ETL-FR3: It should be possible to upload or place a data specification pertaining to specific radar data types (on)to the system.

A web interface has been implemented in order to simplify user interaction with the system. Users have the option to create a data specification for a specific sensor data type. In the case of Apache Storm, this new specification can be used whenever the topology is restarted. In the case of Spark and Hive, this new specification can be used on-the-fly.

In Sections 5.1.2 and 5.4, the setup for the creation of a DDS is discussed. Through the created web interface, it is possible for users to specify message contents for a specific sensor type. This specification is used to validate incoming data during the ETL process.

ETL-FR4: The system should check incoming data against the relevant data specification.

Storm performs this functionality automatically by setting the fields in a specification as required output fields for a topology stream. For Spark and Hive, validators have been built in that check incoming data structures against the specification.

ETL-FR5: The system should reject any data that does not match the specification.

Again, Storm performs this functionality automatically. When the fields in an incoming data structure do not match any of the required fields of an output stream, the data will be rejected. In Spark and Hive, the built validators will decide whether data may be ingested into HBase.

6.2.3 Querying

QUE-FR1: It should be possible for a user to create complex queries to extract data from the cluster.

With Apache Spark as an underlying processing engine, a querying system has been realized. With the querying system, users can define queries that retrieve data from Apache HBase. These queries can contain sets of column qualifiers to retrieve data from, as well as filters that further narrow down the result set.

QUE-FR2: A user should be able to query single data source types without any sort of conversion being needed.

Conversion is not applied when data from a single column family (i.e. sensor data type) is queried.

QUE-FR3: A user should be able to query different data source types at once.

Users can query from any of the column families at once, and can also combine them within a query. The system has been made adaptable in order to accommodate for this.

QUE-FR4: In the case where multiple data source types are queried, and some columns have a Convertible relation towards each other, they should be converted in order to get an Equality relation.

In the case that a conversion is necessary in order to get a uniform result set for a query, conversions are loaded from the MySQL database and applied to the data. Fields that are not uniform are converted to a field from a pre-defined base specification.

QUE-FR5: All related fields in the result set of a query should have Equality relations towards each other.

As a result of the above operations, query outputs should always be uniform, provided that the conversions and base specification fields have been defined by the end user beforehand.

6.2.4 Adaptability

ADA-FR1: End users should be able to define a base specification that all data source types can convert to.

In the previously mentioned web interface, users can define a base specification that may be used to define relations and conversions from existing sensor data type fields.

ADA-FR2: It should be possible for end users of the system to define conversions between ambiguous fields and a relevant base specification.

The same web interface allows end users to define relations between a sensor data type field and a base specification field. The three things that are necessary for such a relation are:

- The sensor data type field
- The base specification field
- Any necessary conversion that needs to be applied

Conversions currently exclusively support numeric conversions.

ADA-FR3: Relations must contain a numerical conversion formula.

As mentioned above, conversions currently only support numeric conversions. In the future, additional conversion types may need to be added.

ADA-FR4: By applying conversion when multiple radar data types are queried, the resulting output should be in a uniform format.

When multiple column families are queried, the system checks whether any of the fields have a relation to each other. When this is the case, these fields are all

converted to the corresponding field in the base specification, in order to ensure that output is always uniform. Please note that this only applies in the case where end users define base specification fields, relations and conversions before querying the data.

6.2.5 Access Control

ACC-FR1: It should be possible to restrict access to (certain parts of the) data on the cluster.

As shown in Section 5.5, HBase Access Control has been implemented in order to require authenticated access to specific parts of the dataset. User permissions can be defined on a table, or even on a column family basis in order to gain access.

ACC-FR2: It should be possible to give (a certain subset of the) end users access to the data, when they provide the correct credentials.

As mentioned above, the HBase Access Control technology used required authentication in addition to correct user permissions in order to gain access to the data.

ACC-FR3: It should be possible to force specific data to be encrypted (should happen during ETL).

Unfortunately, the native HBase encryption (HFile encryption) did not match the requirements well enough in order for it to be viable for use. A potential solution is to encrypt the data during ETL via the CP-ABE algorithm. In this way, it would be possible to encrypt data at a cell, column or column family level.

6.3 Experiments

This section will describe the results of several experiments that were conducted, in order to better argue for the use of certain technologies in the prototypes. In this section, two parts of the overall architecture have been subjected to performance experiments. These two parts are ETL prototypes and querying prototypes. Each of the designed prototypes have been tested with various input parameters and subsequently compared in terms of performance.

6.3.1 ETL performance experiment

This section will describe a performance experiment to be executed with the implemented ETL prototypes. The experiment will serve to test performance in terms of data throughput.

6.3.1.1 Apache Storm

The experiment with Apache Storm consisted of the following steps:

1. Submit the Storm topology to the cluster with the following command:

```
$ storm jar topology.jar org.nl.thales.
    ↪ MultiStreamIngestionTopology [args]
```

2. Start committing data to the Kafka queue:

```
$ java -jar kafka-producer.jar test-data.json [kafka
    ↪ topic]
```

3. Check HBase to make sure the data is correctly stored.

4. Verify the time required to process each data entry and determine data throughput.

6.3.1.2 Apache Spark

Spark works with the spark-hbase-connector package, allowing a Spark job to perform HBase API operations. The result is that it is possible to perform ETL of one or multiple files through Spark:

```
$ spark-submit --class SparkETL --master yarn spark-etl.jar [HBase
    ↪ Table/Kafka Topic]
```

The data throughput results are then measured by checking how long it takes for the amount of messages in the Kafka queue to reach zero.

6.3.1.3 Results

The results of the performance comparison experiment for ETL were as follows:

Technology	Time to complete ingestion
Spark Streaming	5 minutes
Storm	8 minutes

Table 6.2: ETL performance experiment results

From these results we can see that Spark can process many records at a faster pace than Storm. There are a few trade-offs that need to be made however: Storm performs at a latency equal to virtually 0 ms, whereas Spark tends to process data in micro-batches, which can have a latency of several seconds. Therefore, Spark is faster, and should be used if a bit of latency is not a problem. If fully real-time processing is necessary, Storm is better. In this case, since data only needs to be stored and latency is not of the essence, even though both technologies are suitable for the demonstrator, Spark Streaming performs slightly better.

6.3.2 Querying performance experiment

In this experiment, query performance has been tested between the prototypes that have been implemented. The experiment consisted of executing a few specific queries on the same dataset, and checking their respective execution times. The prototypes that are compared are the Spark RDD, Spark DataFrame and Spark DataSet API prototypes.

The experiments were performed with three different types of queries:

- Query Type 1: Scan of the full HBase table and storing the result in JSON format in HDFS.
- Query Type 2: Simple filter query, querying data from rows based on a single conditional expression.
- Query Type 3: Complex query, with varying conditional expressions, grouping and sorting.

The collection of query results can be found in the table below.

	Query Type 1	Query Type 2	Query Type 3
Spark RDD	90 seconds	100 seconds	130 seconds
Spark Dataset	50 seconds	55 seconds	67 seconds
Spark DataFrame	48 seconds	52 seconds	66 seconds

Table 6.3: Query performance experiment results

The result of the experiment was to be expected: The Dataset/DataFrame API from the newer Spark SQL libraries outperform the older RDD API by a significant margin. This is likely due to improvements and optimizations in the execution engine. Spark SQL uses an optimized execution engine called Catalyst. On top of that, the general execution engine of Spark, named Tungsten, has been implemented since version 1.5 to improve performance across all Spark operations.

Chapter 7

Conclusions

In this chapter, the results of the project and answer the research question posed at the start of the project will be reviewed. The limitations of the final solution will be shown, as well as any future work that might help improve the overall solution, and make this demonstrator go from a working concept to a viable solution to use in a production environment.

With the implementation of the demonstrator, we can give an answer to the main question of the project: *What is a way to develop a big data system in which sensor data can be stored and uniformly queried, and which technologies are most suitable to perform the storage and querying of this data?*

In order to fully answer this question, it is important to iterate the fact that the main focus of this project was to prove the capabilities of the designed system to perform required functionalities, whereas any additional research into quality metrics such as performance were less important to Thales for the time being.

To answer the research question, each individual subsystem will be considered. If each of these subsystems fulfills the domain-specific requirements set for them, then the overall system will perform the required functionality, and the answer to the research question will be positive.

A design for the data model has been introduced. The used technologies for this are HBase for sensor data combined with a MySQL database for auxiliary support data. Sensor data is stored in a central big HBase table. Users can use the MySQL database to add DDSs to the system, which allows new data to be ingested into HBase by the ETL component. They can also use the MySQL database to add relation specifications to the system, which makes it so data can be cross-correlated between different data sources.

The performed research shows a way in which ETL can be performed. By using Apache Kafka as a data queue and Apache Storm as a data processing engine, data can be ingested from any source into the system via the streaming data processing paradigm. This thesis showcases an ingestion topology in Storm which is able to

validate inbound data according to a pre-defined specification, and persisting valid data in HBase.

Apache Spark has been implemented as a query processing engine, allowing users to retrieve data from the big sensor data table in HBase. This engine has been implemented with the capability of using a user-defined relation specification to create a uniform query output, i.e. a query result in which all data related to each other is represented in the same way.

As mentioned, users have the ability to define a relation specification. By defining conversions between a specific data field and a common standard, it becomes possible to unify related data output for all formats. he way in which users can create relation specifications is described. Additionally, some of the specifics as to how these relations can be designed by users, and how conversions between data formats are made possible are highlighted.

With all of these descriptions listed above, a demonstrator for the project has been successfully implemented. In addition to verifying the fulfillment of the functional requirements, several experiments have been conducted with the help of a simulated data set. These experiments served to distinguish between different technologies that were used to perform ETL and querying in terms of performance.

For the ETL technologies, it became clear that although Spark Streaming performs better than Storm in terms of raw data throughput, there is also a case to be made for the usage of Storm. This is because of the fact that Storm has a far lower latency than Spark Streaming, and also because Storm is inherently more modular to develop for.

For querying, the relatively new Spark Dataset and DataFrame APIs outperformed the older Spark RDD API. This was expected, as the former two APIs make use of an improved job execution engine called Catalyst.

There is still much work to be done before a system like this is fully production-ready, but the demonstrator does prove that the development and integration of such a system is viable. Section 7.1 will highlight the limitations of the current demonstrator and why these limitations exist, as well as the possible applications of the created design in other fields. Solutions that could potentially solve them will be listed as future work in Section 7.2.

7.1 Discussion

In this section, we will discuss the limitations of the demonstrator that has been implemented. With a limited amount of time, it is obvious that not every aspect of the required system can be highlighted, and many improvements can still be made. To identify these improvements, a critical look at the demonstrator must be taken.

Scalability of the chosen solution has been intentionally overlooked in favor of testing functional requirements of the system via a demonstrator. The approach taken worked on the assumption that the chosen technologies solve the scalability problem by employing cluster computing and related software to provide (near)-linear scalability. In future research, it might be nice to research the actuality of this claim for the purpose of this application of big data.

Due to time limitations, only a few technologies have been practically researched and integrated into the demonstrator. There are quite a few projects available in the Apache Hadoop software ecosystem that provide similar functionalities, and thus are able to fulfill similar functions. It is possible that certain technologies could perform better than the ones used to build the demonstrator. Researching this would be considered a part of future work.

Finally, to return to the mention of different classes of companies benefiting from the proposed design, it can be argued that the implemented system can work for many different classes of companies. As some of the works mentioned in Section 2.3 imply, sensor data can come from many different types of machinery, and combining this data is generally considered a challenge.

7.2 Future Work

There are many different approaches possible when it comes to setting up a big data cluster. During this project, a few approaches have been researched and implemented into a demonstrator. In the future, Thales could potentially look into different technologies that can perform better on different fields, such as data storage, ETL or ingestion and query processing.

During this project, security took a background to the capabilities of the system in terms of importance. As a result, there is still work to be done in this particular field when it comes to big data. Future work in this regard could include the research into technologies that fully satisfy Thales security requirements. These new technologies should also have their impact on performance and other potential quality metrics tested.

In the long term, the system should be prepared to be ready for production. There are still many things that need to be done before the system can be properly used. Some of the main work that still needs doing includes testing the software on a production-size cluster, optimization of the prototype software and further exploring technologies within the big data ecosystem.

References

[1] dictionary.com. “Dictionary.com Unabridged.” In: (Oct. 2017). URL: http://www.dictionary.com/browse/adaptability.

[2] Oxford Dictionaries. “pattern recognition — Definition of pattern recognition in English by Oxford Dictionaries.” In: (Oct. 2017). URL: https://en.oxforddictionaries.com/definition/pattern_recognition.

[3] Simon Kingsley and Shaun Quegan. *Understanding radar systems*. Vol. 2. SciTech Publishing, 1999.

[4] Narayan S Umanath and Richard W Scamell. *Data modeling and database design*. Nelson Education, 2014.

[5] Prajwol Sangat, Maria Indrawan-Santiago, and David Taniar. “Sensor data management in the cloud: Data storage, data ingestion, and data retrieval.” In: *Concurrency and Computation: Practice and Experience* 30.1 (). e4354 cpe.4354, e4354. DOI: 10.1002/cpe.4354. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4354. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4354.

[6] Y. Hajjaji and I. R. Farah. “Performance investigation of selected NoSQL databases for massive remote sensing image data storage.” English. In: *2018 4th International Conference on Advanced Technologies for Signal and Image Processing, ATSIP 2018*. 2018, pp. 1–6. URL: www.scopus.com.

[7] G. Manogaran et al. “A new architecture of Internet of Things and big data ecosystem for secured smart healthcare monitoring and alerting system.” English. In: *Future Generation Computer Systems* 82 (2018). Cited By :7, pp. 375–387. URL: www.scopus.com.

[8] L. Luo. “Data Acquisition and Analysis of Smart Campus Based on Wireless Sensor.” English. In: *Wireless Personal Communications* 102.4 (2018), pp. 2897–2911. URL: www.scopus.com.

[9] B. Shen et al. “A method of HBase multi-conditional query for ubiquitous sensing applications.” English. In: *Sensors (Switzerland)* 18.9 (2018). URL: www.scopus.com.

[10] S. -. S. Fan et al. “Using machine learning and big data approaches to predict travel time based on historical and real-time data from Taiwan electronic toll collection.” English. In: *Soft Computing* 22.17 (2018). Cited By :2, pp. 5707–5718. URL: www.scopus.com.

[11] B. Tekinerdogan. “Synthesis-Based Software Architecture Design.” University of Twente, Mar. 2000. ISBN: 90-365-1430-4.

[12] C.L. Philip Chen and Chun-Yang Zhang. “Data-intensive applications, challenges, techniques and technologies: A survey on Big Data.” In: *Information Sciences* 275.Supplement C (2014), pp. 314–347. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2014.01.015. URL: http://www.sciencedirect.com/science/article/pii/S0020025514000346.

[13] Gordon E Moore. “Cramming more components onto integrated circuits.” In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[14] H. V. Jagadish et al. “Big Data and Its Technical Challenges.” In: *Commun. ACM* 57.7 (July 2014), pp. 86–94. ISSN: 0001-0782. DOI: 10.1145/2611567. URL: http://doi.acm.org/10.1145/2611567.

[15] Du Zhang. “Granularities and inconsistencies in big data analysis.” In: *International Journal of Software Engineering and Knowledge Engineering* 23.06 (2013), pp. 887–893.

[16] T Nasser and RS Tariq. “Big data challenges.” In: *J Comput Eng Inf Technol 4: 3. doi: http://dx. doi. org/10.4172/2324* 9307.2 (2015).

[17] Apache Foundation. *NameNode - Hadoop Wiki*. Apache Foundation. 2011. URL: https://wiki.apache.org/hadoop/NameNode (visited on 11/06/2017).

[18] Vinod Kumar Vavilapalli. *Apache Hadoop YARN - ResourceManager - HortonWorks*. HortonWorks. 2012. URL: https://wiki.apache.org/hadoop/NameNode (visited on 11/06/2017).

[19] Apache Foundation. *JobTracker - Hadoop Wiki*. Apache Foundation. 2010. URL: https://wiki.apache.org/hadoop/JobTracker (visited on 11/06/2017).

[20] Apache Foundation. *TaskTracker - Hadoop Wiki*. Apache Foundation. 2009. URL: https://wiki.apache.org/hadoop/TaskTracker (visited on 11/06/2017).

[21] Apache Foundation. *DataNode - Hadoop Wiki*. Apache Foundation. 2011. URL: https://wiki.apache.org/hadoop/DataNode (visited on 11/06/2017).

[22] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://ezproxy2.utwente.nl:3137/10.1145/1327452.1327492.

[23] Gao, C. *The Overall MapReduce Word Count Process*. [Online; accessed November 14, 2018]. 2017. URL: https://wikis.nyu.edu/download/attachments/86051249/WordCount%20MapReduce%20Paradigm.PNG?version=1&modificationDate=1500920287746&api=v2.

[24] Rick Cattell. "Scalable SQL and NoSQL data stores." In: *Acm Sigmod Record* 39.4 (2011), pp. 12–27.

[25] Dhruba Borthakur. *HDFS Architecture Guide*. Aug. 2013. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[26] Apache Foundation. *Apache Hive TM*. Apache Foundation. 2017. URL: https://hive.apache.org (visited on 10/25/2017).

[27] Apache Foundation. *Apache Cassandra*. Apache Foundation. 2017. URL: http://cassandra.apache.org (visited on 10/26/2017).

[28] Apache Foundation. *Apache HBase - Apache HBase TM Home*. Apache Foundation. 2017. URL: https://hbase.apache.org (visited on 10/25/2017).

[29] Mike Fleckenstein and Lorraine Fellows. "Modern Data Strategy." In: *Modern Data Strategy*. Springer, 2018, p. 240.

[30] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. "Column-oriented database systems." In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1664–1665.

[31] Jeffery L Teeters et al. "Neurodata without borders: creating a common data format for neurophysiology." In: *Neuron* 88.4 (2015), pp. 629–634.

[32] Nan C. Shu, Barron C. Housel, and Vincent Y. Lum. "CONVERT: A High Level Translation Definition Language for Data Conversion." In: *Commun. ACM* 18.10 (Oct. 1975), pp. 557–567. ISSN: 0001-0782. DOI: 10.1145/361020.361023. URL: http://doi.acm.org/10.1145/361020.361023.

[33] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. "A proposed model for data warehouse ETL processes." In: *Journal of King Saud University-Computer and Information Sciences* 23.2 (2011), pp. 91–104.

[34] Octavian Patrascoiu. "Yatl: Yet another transformation language-reference manual version 1.0." In: (2004).

[35] Tova Milo and Sagit Zohar. "Using schema matching to simplify heterogeneous data translation." In: *vldb*. Vol. 98. 1998, pp. 24–27.

[36] Ravi S Sandhu et al. "Role-based access control models." In: *Computer* 29.2 (1996), pp. 38–47.

[37] J. Bethencourt, A. Sahai, and B. Waters. “Ciphertext-Policy Attribute-Based Encryption.” In: *2007 IEEE Symposium on Security and Privacy (SP ’07)*. May 2007, pp. 321–334. DOI: 10.1109/SP.2007.11.

[38] Apache Foundation. *Apache Tez - Welcome to Apache Tez TM*. Apache Foundation. 2017. URL: https://tez.apache.org (visited on 10/25/2017).

[39] Apache Foundation. *Apache Spark TM - Lightning-Fast Cluster Computing*. Apache Foundation. 2017. URL: https://spark.apache.org (visited on 10/25/2017).

[40] Apache Foundation. *Impala - The Apache Software Foundation!* Apache Foundation. 2017. URL: https://impala.apache.org (visited on 10/25/2017).

[41] Apache Foundation. *Apache Phoenix: Overview*. Apache Foundation. 2017. URL: https://phoenix.apache.org (visited on 10/25/2017).

[42] Jacek Laskowski. *DAGScheduler — Stage-Oriented Scheduler*. DataBricks. 2018. URL: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dagscheduler.html (visited on 09/19/2018).

[43] R. Hu, Z. Wang, W. Fan and S. Agarwal. *Cost Based Optimizer in Apache Spark 2.2*. DataBricks. Aug. 31, 2017. URL: https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html (visited on 09/19/2018).

[44] Zhang, Zhan. *Spark-on-HBase: DataFrame based HBase connector - Hortonworks*. HortonWorks. 2016. URL: https://hortonworks.com/blog/spark-hbase-dataframe-based-hbase-connector/ (visited on 10/11/2018).

[45] Damji, Jules. *RDD vs DataFrames and Datasets: A Tale of Three Apache Spark APIs*. Databricks. 2016. URL: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html (visited on 10/11/2018).

[46] Apache Foundation. *8.2. Access Control*. Apache Foundation. 2018. URL: https://hbase.apache.org/0.94/book/hbase.accesscontrol.configuration.html (visited on 09/11/2018).

Appendix A

Collection of Hadoop-supporting Apache projects

Name	Category	Prevalence	Short description
Hadoop Common	Core	Always	Also known as Hadoop Core, it contains all the common utilities and libraries that are used to support other Hadoop modules.
Hadoop Distributed File System (HDFS)	Core	Always	Distributed file system designed to run on commodity hardware. Used for storing large volumes of data across a cluster of computers.
Hadoop YARN (Yet Another Resource Negotiator)	Core	Always	Cluster management technology. It is the architectural center of Hadoop. It enables several data analysis technologies, such as real-time streaming and batch processing.
Hadoop MapReduce	Core	Always	Software framework that allows a user to quickly write software that processes vast amounts of data in-parallel on large clusters. Guarantees reliability and fault-tolerance.
Avro	Framework	Uncommon	Data serialization framework. Allows for the representation of complex structures in MapReduce jobs.
Oozie	Framework	Common	Job scheduler for Apache Hadoop. Support for Java MapReduce, streaming MapReduce, Pig, Hive and Sqoop.

Slider	Framework (Incubating)	Uncommon	Deployment engine for deploying existing distributed applications onto a Hadoop cluster. The tool itself is a Java CLI program, which persists information it receives on HDFS as JSON files.
Spark	Framework	Common	Computing framework for computer clusters. Requires a cluster manager (standalone Spark, Hadoop YARN or Apache Mesos) and a distributed storage system (HDFS, MapR-FS, Cassandra, OpenStack Swift, Amazon S3 or Kudu, or any custom implementation)
Storm	Framework	Common	Distributed stream processing computation framework written mostly in Clojure.
Tez	Framework	Uncommon	Framework for building high-performance batch processing applications. Dramatically increased processing speed as opposed to MapReduce, while maintaining data scalability.
ZooKeeper	Framework	Common	Distributed hierarchical key-value store, used as a distributed configuration service, a synchronization service and a naming registry for large distributed systems. Use cases: Naming service, config management, sync, leader election, message queue and notification system.
MapR DB	Framework	Uncommon	NoSQL DBMS. Contains the HBase API so that it can run HBase applications at higher performance.
Myriad	Framework (Incubating)	Uncommon	Myriad allows for the deployment of Hadoop YARN applications on Apache Mesos, essentially making it so both are implicitly present on the system.
Accumulo	NoSQL	Uncommon	Sorted distributed key-value store, based on Google's Bigtable. Third most popular NoSQL engine behind Cassandra and HBase.

HBase	NoSQL	Common	Official Hadoop database. Provides real-time read/write access. Provides Bigtable-like functionlity on top of the Hadoop framework and HDFS.
Parquet	NoSQL	Uncommon	Column-oriented data store on Hadoop.
Hive	SQL	Common	SQL-like interface for querying data in Hadoop. Can run as MapReduce jobs to run queries in a distributed fashion.
Impala	SQL (Incubating)	Uncommon	Formerly Cloudera Impala. It is an SQL engine that can massively process queries in parallel.
Phoenix	SQL	Uncommon	Relational database engine for HBase. Compiles the user's SQL queries to native NoSQL store APIs.
Atlas	Data Governance	Uncommon	
Calcite	Data Governance	Uncommon	SQL parser, an API for building expressions in relational algebra and a query planning engine.
Falcon	Data Governance	Uncommon	
Crunch	Abstraction	Uncommon	Framework for writing, testing and running MapReduce pipelines. Runs on top of MapReduce and Spark. It is a Java API for joining and aggregating data.
Pig	Abstraction	Common	Platform for creating programs that run on Hadoop. Utilizes a high-level languages called Pig Latin (similar to SQL), effectively turning MapReduce jobs into a high-level programming paradigm. These jobs may be run on MapReduce, Apache Tez and Apache Spark.

DataFu	Data Mining (Incubating)	Uncommon	Collection of libraries for working with large-scale data mining. Consists of a library of User Defined Functions (UDFs) for Pig, and an incremental processing framework in MapReduce called Hourglass.
Flume	Ingestion	Common	A distributed, reliable and available service for efficiently collecting, aggregating and moving large amounts of log data.
Kafka	Ingestion	Uncommon	Distributed stream processing framework, complementary to Storm, Spark and HBase.
Streams	Ingestion (Incubating)	Uncommon	Contains libraries and patterns for specifying, publishing and inter-linking schemas, and assists with conversion of activities (posts, shares, likes, follows, etc.) and objects (profiles, pages, photos, videos, etc.) between the representation, format and encoding preferred by supported data providers (Twitter, Instagram, etc.) and storage services (Cassandra, Elasticsearch, HBase, HDFS, Neo4J, etc.)
Sqoop/Sqoop2	Ingestion	Common	A tool designed for efficiently moving bulk data between Hadoop and structured datastores (e.g. relational databases).
Knox	Security	Uncommon	A system that provides a single point of authentication and access for Apache Hadoop services in a cluster.
Ranger	Security	Uncommon	A framework to enable, monitor and manage comprehensive data security across the Hadoop platform.
Mahout	Machine Learning	Common	A simple and extensible programming environment and framework for building scalable machine learning algorithms.
Hue	Interface	Uncommon	A smart Analytics Workbench. A web interface that may be used to analyze data from Hadoop.

Solr	Inter-face	Uncom-mon	Solr (pronounced Solar) is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene.
Lucene	x	x	x
Ambari	x	x	x
Cassandra	x	x	x

Appendix B

Converter Function

```
public static byte[] convert(String key, byte[] value){
    String valStr = Bytes.toString(value);

    String query = "SELECT␣conversion_to_base_type␣" +
            "FROM␣relations␣WHERE␣column_qualifier_id=(SELECT␣id␣
            ↪ FROM␣column_qualifiers␣WHERE␣full_qualifier_name
            ↪ =?)";

    try(Connection con = DriverManager.getConnection(url, user,
        ↪ password)){
        PreparedStatement pst = con.prepareStatement(query);

        pst.setString(1, key);
        ResultSet rs = pst.executeQuery();

        if(rs.next()){
            String conversionStr = valStr + rs.getString(1);

            ScriptEngineManager manager = new ScriptEngineManager
            ↪ ();
            ScriptEngine engine = manager.getEngineByName("
            ↪ javascript");
            Object result = engine.eval(conversionStr);

            return Bytes.toBytes(result.toString());
        }
    }
    catch(Exception ex){
        ex.printStackTrace();
    }

    return value;
}
```