Utrecht University

Master Thesis

Efficiently handling data schema changes in an event sourced system

Author:
Marten Spoor

First Supervisor:
dr. Slinger Jansen
Second Supervisor:
drs. Hans Philippi
External Supervisor:
Michiel Overeem, MSc.

August 29, 2016

Universiteit Utrecht

Efficiently handling data schema changes in an event sourced system

Marten Spoor

Abstract

Data schema changes occur as a result of evolving software systems. This is a well-known problem and for relational data models there is a large body of research that addresses this problem. However, this research does not apply to event sourced systems, because in those systems there is no relational model and often there is no explicit data schema. A second problem is that in event sourced system the amount of data is much larger than in conventional systems, because the system does not only store the current state but every event that changed the state.
In this research, a framework is created to illustrate which schema changes are to be expected, reflected by operations, executed with data transformation techniques and deployed according to a deployment strategies to execute these operations. When creating the framework, efficiency including zero downtime was. When completed, the framework was evaluated with three Dutch experts. The final framework provides a starting point for developers that are currently building an event sourced system and that are searching for a solution for the problems of the changing data schema.

Keywords: *Software evolution, event sourcing, data schema, implicit schema, operations, data transformation techniques, deployment strategies, zero downtime, event store*

Acknowledgments

This thesis is the final chapter of my master Business Informatics and my student life. During my student life I learned a lot, met many new cool people, and made a lot of new friends.

When writing this thesis, various people helped me in some way, which all deserve a thank you. First of all thanks to Utrecht University supervisors Slinger Jansen and Hans Philippi from Utrecht University, for the help and the useful feedback. Next, I would like to thank my daily supervisor at AFAS, Michiel Overeem for all the pleasant brainstorms, meetings, and fine collaboration. I also want to thank some others at AFAS: Rolf de Jong and Machiel de Graaf for giving me the opportunity to do my master thesis at AFAS Software and the backend team for all the help and funny lunches. Thanks to the experts in event sourcing, Pieter Joost van de Sande, Allard Buijze and Dennis Doomen for the interviews and useful feedback.

I would like to thank Sander Klock, Pepijn Gramberg and later on Bart Smolders for all the pleasant foosball games and helping each other during our graduation projects. Thanks to my friends, Rick Barneveld and Peter Boot, for proofreading and providing me with good feedback, even during their vacation.

And finally, a special thanks to my parents and sister, for all their advice and support, not only during this project, but during my whole (student) life.

Marten Spoor

Contents

1 Introduction **2**
- 1.1 Problem statement . . . 3
- 1.2 Research questions . . . 4
- 1.3 Research relevance . . . 6
- 1.4 Thesis overview . . . 6

2 Research approach **8**
- 2.1 Design science . . . 8
- 2.2 Literature study . . . 9
- 2.3 Evaluation with experts . . . 9
- 2.4 Research context . . . 10

3 CQRS and event sourcing **12**
- 3.1 Introduction to CQRS and event sourcing . . . 12
- 3.2 CAP and eventually consistent . . . 14
- 3.3 Domain-driven design and model-driven development . . . 15
- 3.4 In-depth architecture . . . 16

4 Related literature **20**
- 4.1 Implicit schema and schemaless . . . 20
- 4.2 Databases and data stores . . . 21
- 4.3 Event-driven architecture & event processing . . . 23

5 Data transformation operations **25**
- 5.1 The structure and schema in the event store . . . 26
- 5.2 The operations . . . 28

6 Techniques for data transformation **43**
- 6.1 Schema evolution versus versioning . . . 43
- 6.2 Techniques for other kinds of databases . . . 45
- 6.3 Event store techniques . . . 48

6.4 Analysis and comparison of different event store techniques . . 53

7 Deployment strategies **57**
7.1 Strategies 57
7.2 Deployment strategies analysis 63

8 The design of an upgrade strategy **65**
8.1 Operations and techniques 65
8.2 Techniques and strategies 66
8.3 Final framework 68

9 Evaluation with experts **72**
9.1 Evaluating the operations overview 72
9.2 Expert interviews about the framework 73

10 Discussion **77**
10.1 Construct validity 77
10.2 Internal validity 77
10.3 External validity 78
10.4 Reliability 78

11 Conclusion **79**
11.1 Main conclusions 79
11.2 Future research 83

Glossary **84**

Bibliography **86**

A Tables techniques & deployment strategies combined **92**

B Explanation tables operations, techniques and strategies **99**

C Expert interview: interview protocol **103**
C.1 Protocol 103
C.2 Preliminary figures used during interviews 105

D Expert interview: summaries **107**
D.1 Allard Buijze 107
D.2 Dennis Doomen 109
D.3 Pieter Joost van de Sande 110

"A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful."

— Meir M. Lehman

Chapter 1

Introduction

Lehman's law *continuing change*, as quoted on the first page, states that applications that do not evolve become less useful (Lehman, 1980). Although the law is almost forty years old, it still holds for new applications, even if they use new emerging architectural patterns and technologies. Therefore the quest for ways to handle evolving application keeps continuing, following each new pattern and technology on its heels.

One of those new patterns is CQRS. In 2009, Greg Young and Udi Dahan introduced CQRS, which stands for *Command Query Responsibility Segregation* (Dahan, 2009; Young, 2010b). CQRS is an architectural pattern that separates the commands (changing state) from the queries (views on the state). Because the commands and queries are handled separately, they can be executed by different components, using different data sources, which makes it possible to optimize both sides.

CQRS is seen as a stepping stone for *event sourcing* (Young, 2016). In *event sourcing*, instead of only saving the current application state, the changes leading up to that state are saved, similar to an audit trail (Fowler, 2005). Because all the steps are saved, you can always completely rebuild the current state by simply executing the saved steps. Using event sourcing has technical benefits like improved debugging. Because you can reproduce the steps that led to the bug. Furthermore, it creates extra business value because you can rebuild the application state to any point in time and you know how the application got there. This is ideal for technologies like business intelligence and pattern recognition, as you can do data analysis on all the events that happened through time. Within CQRS, event sourcing is often used as the data source for the command side of the application.

According to Lehman's law, you need to keep evolving your application to stay useful and relevant. The data schema describes the structure of the data inside an application. This can be done in a formal language and explicit, or it can be implicit, only derivable by analyzing the source code. The data schema is one of the aspects of an application that undergoes evolution. In an event sourced application, managing and applying these changes from the perspective of a developer can be challenging. From the perspective of the user there is another major challenge challenge: can the evolution be applied in a manner that will not bother the user. These challenges form the motive for this research. The next sections will give an overview of the research questions that will be answered.

1.1 Problem statement

As the application evolves, data schema changes can be triggered, requiring a transformation of the existing data. This is challenging for applications using event sourcing because these implementations do not use an explicit data schema, on which existing techniques and strategies from the research on relational databases can be applied.

While an event sourced application may not have an explicit data schema, it does not mean that the data is schemaless. All though there is no formal way to describe the schema of the events, knowledge about the structure and types of the events is embedded in the application. This is called an implicit schema: there is no formal schema that describes the structure of the data, the application assumes a certain structure (Fowler, 2013b). Although the implicit schema can be found in the application, the data in the data store (in the case of event sourcing the event store) should comply to the implicit schema.

Not only having an implicit schema is challenging, an event sourced application consist of substantially more data compared to a relational approach. A relational approach only saves the last state of the data, in an event sourced approach every change is saved as an event, causing substantially more data.

Current knowledge is lacking about what to expect and how to handle these expected changes (Fowler, 2016). There are existing techniques for doing data transformation in an event sourced application, but there are no sources that give guidance in the design and the rationale of the upgrade

as a result of a schema change. These existing techniques also lack the characterization regarding quality attributes that are required.

The contribution of this research is a framework that guides the upgrading of an event sourced application, by making the data transformation operations explicit and giving a rationale for choosing the appropriate technique and deployment strategy.

1.2 Research questions

Based on the problem statement and the focus of this research project, we define the main research question as:

RQ: *"How can an event sourced application efficiently be upgraded in the face of event schema changes?"*

An event schema change is triggered by an evolving application. We define *event schema* as the implicit data schema of events in an event store. Several steps are important when executing the upgrade as a result of a schema change. These steps are depicted in Figure 1.1. The schema changes are reflected by *operations*, which need to be performed by *techniques* and deployed according to a *strategy*. When the steps that should be taken are clear, the evolution of application V1 to application V2 can be executed.

The following subquestions are defined to answer the main research question:

SRQ 1: *'Which operations are needed to transform events so that they comply to the new event schema?"*

The research will start with creating an overview which operations can be expected during the upgrade of an event sourced application and its implicit data schema. These operations reflect the changes done in the implicit event schema and should transform the data in such way that the events reflect the new event schema.

SRQ 2: *"Which techniques are available to execute data transformation operations?"*

Different techniques already exist for data transformation on all kinds of

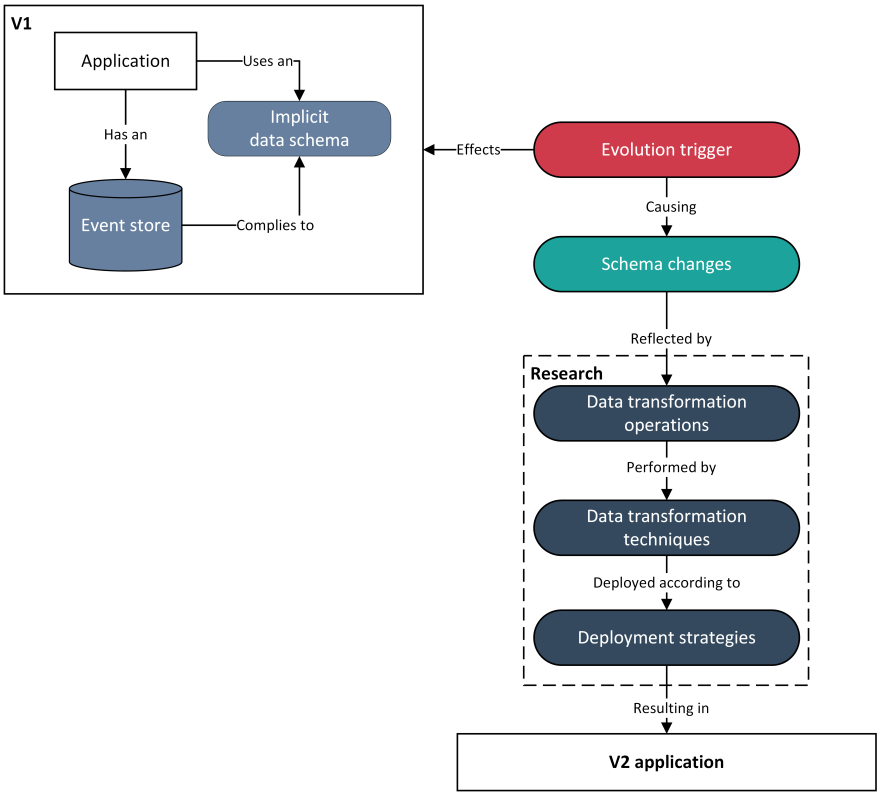

Figure 1.1: Data transformation of an event sourced application and the position of this research.

databases and data models. Therefore, with the help of existing techniques, an overview of the event store techniques will be created.

SRQ 3: *"What are the existing deployment strategies for upgrading software?"*

Besides knowledge of what can be expected and possible execution techniques, an overview of the existing deployment strategies is needed. These strategies are approaches to get the techniques deployed and/or working. When looking for deployment strategies the efficiency factor will be kept in mind, this because the amount of data expected is higher than in traditional databases.

SRQ 4: *"How can the most appropriate technique be selected and deployed efficiently, given a set data transformation operations?"*

The results of subquestion 1, 2 and 3, being the operations, techniques and strategies, need to be designed and combined: what is the most appropriate technique for a set of operations, to be deployed and achieve an efficient

upgrade. *Efficient* here does not only mean zero downtime but also that it is efficient in performance. The reason to add the latter is that an application upgrade with zero downtime that takes multiple days is not considered efficient. This not only a matter of combinations but also about designing, to chose that appropriate technique based on different aspects.

1.3 Research relevance

This research is relevant because of several reasons:

Create a scientific base for CQRS & event sourcing - as CQRS and event sourcing is relatively new in the scientific world, only a few papers are published related to these subjects. This research will help in creating a better scientific base and hopefully interest from the scientific world, to connect more with this topic on the current development in software development world.

Upgradability of event sourcing application - being one of the bigger challenges in such applications, many software companies struggle with finding the right approach for solving these problems. These software companies can benefit from the framework which will be created. The framework can help them making decisions regarding their event sourced application and how they want to upgrade it.

AMUSE - this research project is a part of a bigger research project, called AMUSE. This research will bring AMUSE further in their goal to design **A**daptable **M**odel-based and **U**ser-specific **S**oftware **E**cosystems. Read more in section 2.4.2.

1.4 Thesis overview

The structure of this thesis is as follows. This chapter is followed by the research approach in Chapter 2. The research approach is followed by an introduction to CQRS & event sourcing (Chapter 3) and other related literature (Chapter 4) for this research. This is followed by the overview of operations (SRQ 1, Chapter 5), existing techniques for both event stores and other types of databases (SRQ 2, Chapter 6) and strategies to deploy them (SRQ 3, Chapter 7). This all is combined in Chapter 8 where the relationships between the operations/techniques/strategies are described, and the

final framework is presented. Parts of this research are evaluated with the help of experts. These evaluations are described in Chapter 9. This thesis ends with discussing the validity of this research in Chapter 10, and the final Chapter 11, which summarizes the results and suggests future work.
When you are looking for the definition of a specific term, you can check the glossary, which is added at the end of the thesis.

Chapter 2

Research approach

In this chapter, the research approach is explained, the performed literature study and expert interviews and the research context, being AFAS and AMUSE.

2.1 Design science

In this research project, the approach from the Information Systems Research Framework by Hevner et al. (2004) is used. This research is projected on the framework in Figure 2.1. This approach combines three different aspects: environment, knowledge base and Information Systems (IS) research.

The environment defines the place where the problem exists and where the Business Needs comes from. The knowledge base is based on existing foundations and methodologies and gives us the Applicable Knowledge in the context of our IS research. The third aspect is IS research, which is our research project, that receives input from the business needs and applicable knowledge and combines that to what we are developing/building and how we are evaluating it. As a result of our IS research, we are making an addition(s) to the knowledge base and are creating something that is applicable in the environment.

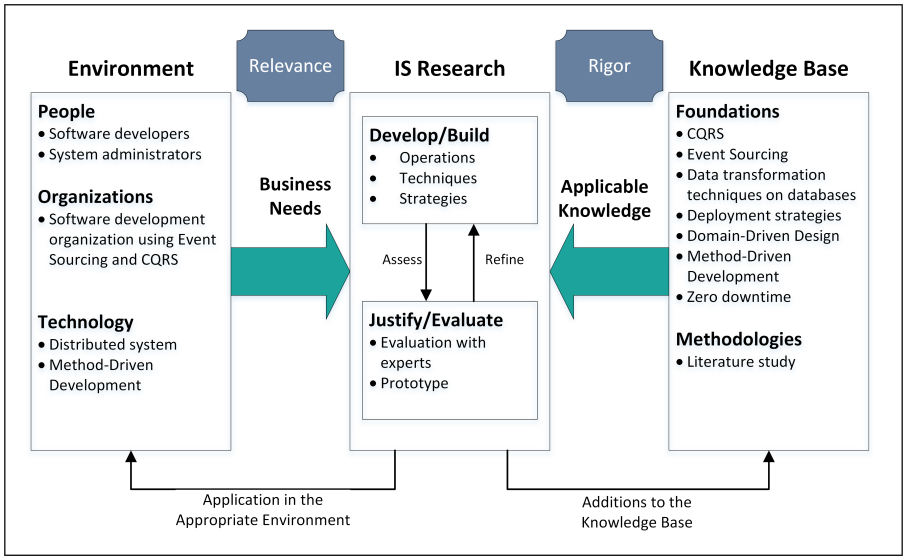

Figure 2.1: Our research project mapped on the Information Systems Research Framework (Hevner et al., 2004)

2.2 Literature study

A literature study is conducted in different phases of this research, all using the snowballing technique. Before working on the research questions, a literature study is conducted to get familiar with CQRS and event sourcing and the context. The result of this context literature study can be found in chapter 3. In chapter 4 the other related literature that was found during this research is discussed. Here, more general concepts which are related to this research are described, like different *data store types* and the *event-driven architecture*. Furthermore, specific literature regarding *schema evolution, schema versioning*, and *data transformation techniques* are outlined in Chapter 6. To answer the third subquestion in Chapter 7, some more literature was researched regarding the topic of *deployment strategies*.

2.3 Evaluation with experts

Two evaluation steps with experts were held during this research to improve the validity and generalizability of this research.

The first evaluation step was performed with the internal experts from AFAS. An evaluation session was held with 3 AFAS experts, which only in-

corporated the operations part (SRQ 1) of this research. This was conducted because the operations were the base for the rest of the research. The second evaluation step was performed by conducting external expert interviews, to see if their experiences match with the research and if they would have any additions or remarks to the framework. The interviews were semi-structured and according to the tips of Jacob and Furgerson (2012). The feedback from the experts was used to improve the different results and the final framework of this thesis.

The results of the evaluation steps can be found throughout the thesis as it is used to improve the results. A description on what changes and insights were identified during the evaluation steps can be found in Chapter 9.

2.4 Research context

This research project was conducted at software company AFAS Software B.V., as part of AMUSE and their Profit Next project. Therefore I will briefly describe AFAS Software, Profit Next and AMUSE.

2.4.1 AFAS Software

AFAS Software B.V. is a Dutch organization focused on building enterprise software. AFAS was founded in 1996 after a management buy-out from Getronics. The headquarters of AFAS is located in Leusden, the Netherlands. The international offices of AFAS Software B.V. are located in Belgium and Curaçao. At the time of writing, AFAS employs over 350 people. The vision of AFAS is to automate administrative business processes. To realize this vision, AFAS focuses on building and retailing an integrated enterprise software product for both small, and large organizations. In 2015, AFAS realized a revenue of 92,3 million euros and a profit of over 30 million euros.

2.4.1.1 Profit Next

AFAS is rebuilding their software product from scratch, calling this project Profit Next[1]. It is a long-term project of which the first release is currently

[1]Dutch introduction and explaining video: http://dev.afas.nl/1337/ontwikkelvisie

planned for 2019. In this project, they have the vision of bringing their application to the cloud, with separation of technique and functionality. Because of this separation, they can quickly switch between different platforms. Furthermore, they are using techniques like model-driven design and continuous deployment on both the customer and developer side.
The data transformation researched in this research project has a important role in Profit Next. Within Profit Next, CQRS & event sourcing are used in combination with model-driven development. As the implementation of this combination, they generate the event sourced application. Users do not want to wait hours until they can continue using Profit Next when the event schema changes. The same applies on the developer-side. If AFAS updates parts of the system and this brings changes to their standard application model, this deployment needs to go as quick and smooth as possible, with the correct data transformation.

2.4.2 AMUSE

This research is part of the AMUSE project. AMUSE stands for *Adaptable Model-based and User-specific Software Ecosystems* and is an academic collaboration in the Netherlands between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software. AMUSE is addressing software composition, configuration, deployment and monitoring of heterogeneous cloud ecosystems through ontological enterprise modeling. The AMUSE project is funded by the NWO. More information can be found on http://www.amuse-project.org/.

Chapter 3

CQRS and event sourcing

This chapter is an introduction to CQRS and event sourcing, to get familiar with the context of this research and the concepts and theory of CQRS & event sourcing. Readers who are already familiar with these concepts can safely skip this chapter.

3.1 Introduction to CQRS and event sourcing

Command Query Responsibility Segregation (CQRS) is based on the Command-Query Separation (CQS) principle, introduced by Meyer (1988). Meyer defined a command as *serving to modify objects* and a query as *to return information about objects*. Anything else done by a command or query he defined as (abstract) side effects, which according to him should not be produced. Alternatively informally stated in his words *"asking a question should not change the answer"*.

CQS was later picked up by Greg Young and Udi Dahan, which combined the CQS principle with Domain-Driven Design and created the CQRS pattern (Dahan, 2009; Young, 2010b). Domain-Driven Design (abbreviated to DDD) is introduced by Evans (2004). DDD is explained in section 3.3. Within the CQRS pattern, the changing state (command) and the views on state (query) are separated in two different subsystems, which both have their data source. The two different subsystems are updated using a message bus for events between them. This bus is often an asynchronous one. Through this bus, events are sent from the command to the query side. The query side can update the queries according to the events that represent changes

to the state. The huge benefit of this approach is that both subsystems, the command side and the query side, can be optimized individually.

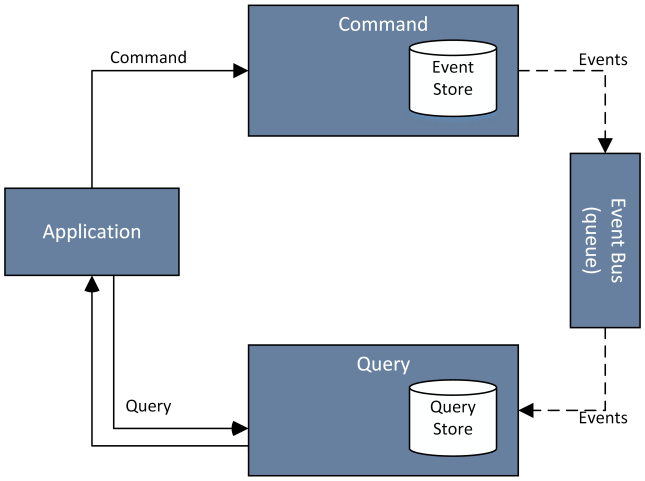

Figure 3.1: Simple CQRS representation

The CQRS pattern is often combined with event sourcing, or as Young (2016) calls it in a recent talk: *"CQRS is a stepping stone to event sourcing"*. For event sourcing, you do not save or update something to the current state like in a normal relational database, but you save all the steps that will lead to the current state, like an audit trail (Fowler, 2005). When all the steps are stored, you can always calculate the current state by simply executing all the steps you stored. When CQRS and event sourcing are combined, event sourcing is used as the schema of the data source on the command side.

Benefits of the combined CQRS & event sourcing approach are that the system becomes better scalable, because of the two systems with an asynchronous link in between. Extra business value is added since you now know everything that happened in your system at any point in time. This provides many business intelligence and pattern recognition options. Furthermore, you can easier debug your system because you can reproduce bugs by replaying the events to the point in time where the bug happened.

Unfortunately, there is little scientific research available on CQRS and event sourcing patterns. Kabbedijk et al. (2012) did a case study on the variability of the CQRS pattern and patterns he refers to as CQRS sub patterns, like event sourcing and command handlers. Korkmaz (2014) described the practitioners view on CQRS and Guelen (2015) wrote about performance testing in the CQRS environment.

3.1.1 Event sourcing example

To illustrate the difference between the relational model approach and the event sourcing approach, a small illustrative example can be found in Figure 3.2. In the example at the left side, the relational approach is depicted and at the right side the event sourcing approach is shown. In the example, a bank account is opened, and some actions are performed on this account. As shown in the relational approach, the entry is just updated at each action. Whereas in the event sourcing approach, not the current state is saved but each time a new event is created, with the delta. With the event sourcing approach, you can replay the events to get the left situation, but you now also know how it got there.

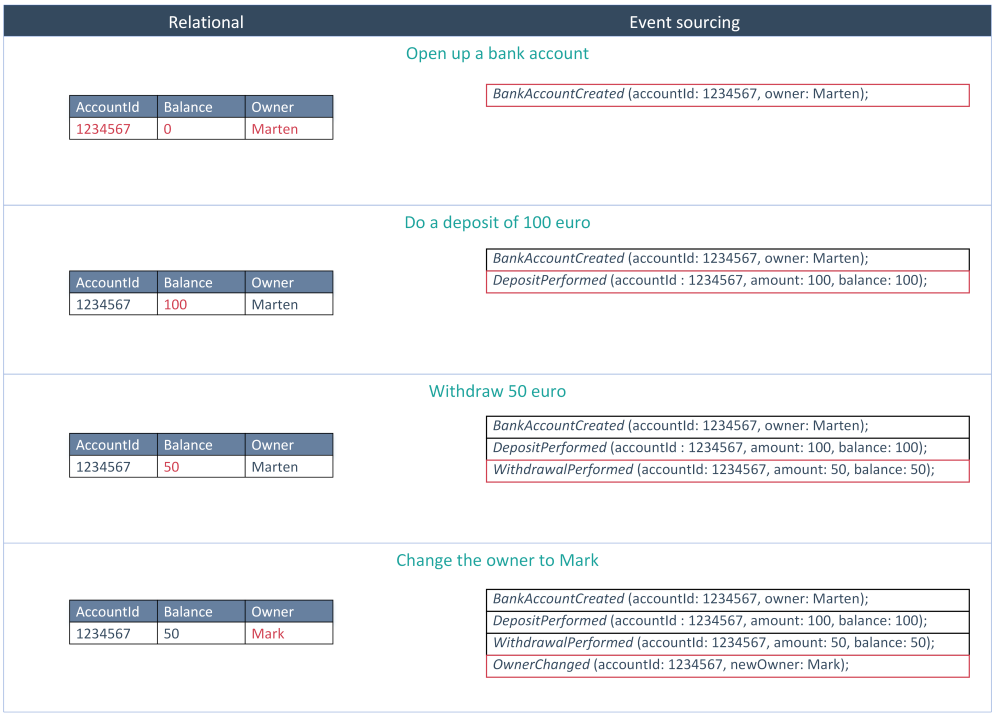

Figure 3.2: Relational vs. event sourcing

3.2 CAP and eventually consistent

The CQRS pattern is interesting to use because it provides a way to get around problems which you will encounter according to the CAP Theorem

(Young, 2010a). CAP stands for Consistency, Availability and Partition tolerance, introduced by Brewer (2000) and later proven by Gilbert and Lynch (2002). The theorem states that in a distributed system you can have at most two out of the three properties Consistency, Availability, and Partition tolerance. More recent publications made that theorem less strict, stating that two out of three is misleading, and a decision between the properties is more continuous than binary (Brewer, 2012). If the consistency property is seen more continuous you (can) end up with forms like eventually consistent. This eventually consistent property is a weaker form of consistency and means that when no updates are made to the object, the object will eventually have the last updated value (Vogels, 2009). Depending on the implementation of the messaging bus between the command side and query side, CQRS can achieve being eventually consistent, although this is not obligated by the CQRS definition. The bus is there to send the new events from the command to the query side of the system so that the query side can update his views on the state. When the bus is an asynchronous implementation, the user does not have to wait until the views are up to date. This improves the scalability and the possibilities to optimize both subsystems. As a downside, it can happen that the views on data return an outdated state of the application, as the event is not yet processed.

3.3 Domain-driven design and model-driven development

As stated, the CQRS pattern includes principles of *Domain-Driven Design* (Evans, 2004) for modeling an application. With Domain-Driven Design (DDD) you describe a system for both IT and business by mapping business domain concepts into software artifacts. Typical concepts which can be found in CQRS and DDD are bounded context and aggregates principles (see section 3.4.1.1).

One of the weaker sides is that DDD has the need for much upfront thought. You first need to think about the model of your domain, before building your application. When the application is running, changes to your domain model can be costly to integrate with the application. Therefore, DDD can clash with development principles like agile development, where you have smaller iterations to show the development progress quickly.

To reduce those problems, Brandolini (2013) introduced an approach to

help modeling an event sourced system called *event storming*. *Event storming* is an approach to brainstorm quickly about the expected aggregates, bounded context and other relevant objects in your application upfront, in a workshop format.

DDD is compatible with Model-driven development (abbreviated MDD). Model-driven development is simply the notation that a model of a system can be transformed into the real thing according to Mellor et al. (2003). MDD is a subset of model-driven engineering and can be described as a development method that uses a model as a principal artifact, to be used in generating an application (Ameller, 2009). When modeling your domain with the help of DDD, the domain model can be used as input for MDD, which together can become a strong combination and is a good fit.

3.4 In-depth architecture

Until now, CQRS and event sourcing architecture were only discussed on a high level. Now components of CQRS will be discussed a bit more in-depth. The in-depth CQRS architecture can be found in Figure 3.3.

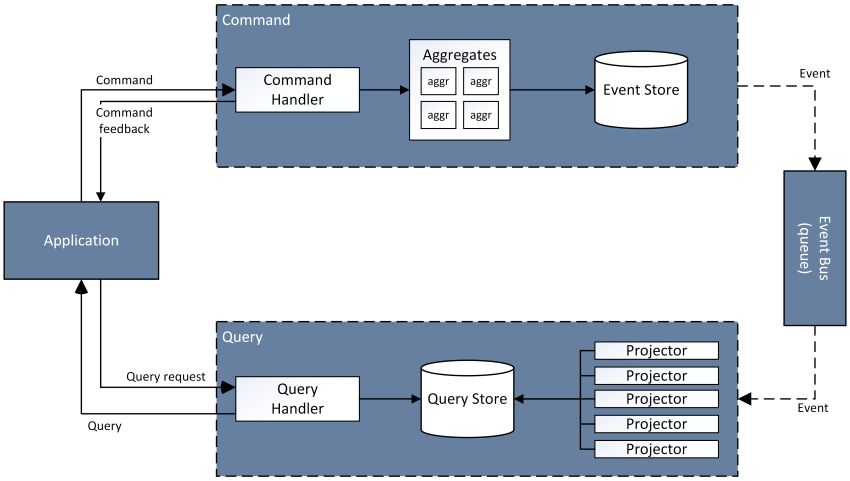

Figure 3.3: CQRS more in depth

3.4.1 Command side

The command side is responsible for validation and processing changes to the application state. It consists of a few key components: being the aggregates, command handlers, and the event store. All will be described in this section.

3.4.1.1 Aggregate

Aggregates are combinations of entities, which combined have their *bounded context*. This means that the entities are related/dependent on each other. The main entity of a bounded context is referred to as *aggregate root*. The aggregate root is responsible for distributing and coordinating the bounded context. The concepts aggregates and bounded context come from Domain-Driven Design (Evans, 2004). These aggregates not only add complexity but provide huge benefits related to scalability. Instead of one big block, the command side can be made of several smaller blocks with their responsibility. These aggregates are responsible for validating commands and when a command is correct, raising a new event to put in the event store and on the event bus.

3.4.1.2 Command handler

The command side consists of several command handlers. Each command is being handled by one command handler. The command is then sent to corresponding aggregate or, when the command invokes multiple aggregates, to the process manager.

3.4.1.3 Event store

When you use event sourcing, the data store that is used, is called event store. In an event sourced application, the event store is immutable. This means events that are stored are never deleted or updated but only inserted or read from the event store. An event store is often implemented with a NoSQL data store, although implementations based on an RDBMS are also possible (Traub and Simmons, 2011). In this case, the event is saved as plain text or JSON/XML. As long as the application can use the data store with an (partial) implicit schema it will work.

As building the state can become quite expensive when there are many events, one of the concepts used to improve the performance is *snapshotting*. When using snapshotting, the application sometimes take a so-called snapshot from the current application state or the state of a specific aggregate, which is saved somewhere in the event store. When rebuilding the application state from the event store with snapshotting, the application do not have to replay all the events from the start. The application can just go a little bit back in time to the latest snapshot and replay the events since that snapshot (see Figure 3.4). More explanation about the inside of the event store can be found in section 5.1.

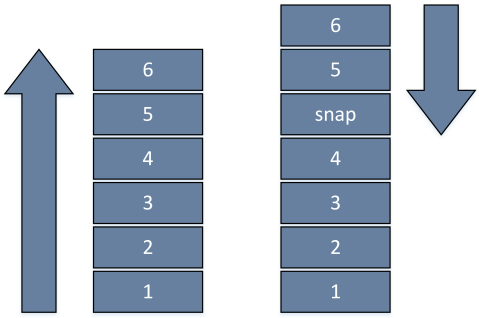

Figure 3.4: Simple view of snapshotting, adapter from Young (2010b)

3.4.2 Query side

As explained earlier, the query side is responsible for views on the state of the application. Just as with the command side, the query side consists of are several concepts. The most important ones being the projector, query handler, and the query store.

3.4.2.1 Projector

The most important concept is *projection*. Projections are views on state or the state history, created and updated by the projector. These are denormalized views, based on which views are required by the application. These projections do not have to be simple data objects, but can also be a PDF or a graph. Not only can it be in any format; the idea behind projections is that they are already prepared for the need of the application. Each time an event is created in the event store, it is sent to the query side of the event

bus. The projector receives the event from the bus and directly builds up the updated projections, so next time it is queried by the query handler it does not have to be calculated. Though, because the event is asynchronous, it can occur that the projection is a little behind because of the eventual consistency of the application. Each projector has a queue mechanism in place. When receiving events, they are put in that queue until the projector has time to process the event.

3.4.2.2 Query handler

The query handler receives query requests from the application. The query handler retrieves the correct query result from the query store, which is then sent to the application.

3.4.2.3 Query store

The query store is the data store for all the queries, created by the projectors. In the query store, the denormalized query result data can be found, which are updated through the projectors and retrieved by the query handlers.

Chapter 4

Related literature

This chapter describes related literature to this research. The related subjects are first described and then their relevance regarding this research subject is explained.

4.1 Implicit schema and schemaless

A schemaless database allows any data, structured with individual fields and structures, to be stored in the database (Fowler, 2013b). This term is used for data stores which do not have a globally defined schema (Scherzinger et al., 2013). Using a schemaless database gives extra flexibility as you do not need to define those schemas. Although your data is stored schemaless, the application still needs to interpret and manipulate the data. Therefore, assumptions about the data are made by the application. The data is said to have an implicit schema (Fowler, 2013b). This schema is not explicitly described/formalized, but the application assumes a certain schema.

The database used in event sourcing, the event store, does not have an explicit schema of what events can occur including which attributes. Event sourcing implementations use an implicit schema, which is based on the types from Section 5.1. An event store is only used to store and replay events, and never to query data. The event store is only for persistence, and therefore does not need to know the event schema. This makes it more flexible because the store and application are loosely coupled. The logic about the schema is completely found in the application where the implicit schema is assumed.

4.2 Databases and data stores

Several databases are considered related to this research, being relational databases, NoSQL stores, and temporal databases.

4.2.1 Relational database

Relational databases are already around some while. Codd (1970) proposed in 1970 a relational model for data, to protect users from creating disruptive changes. The approach, used for data transformation on a relational database application, is in the scientific literature often related to the terminology of *schema versioning* or *schema evolution*. *Schema versioning* is accommodated when a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces. *Schema evolution* is accommodated when a database system facilitates the modification of the database schema without loss of existing data. More regarding schema versioning and evolution and existing data transformation techniques, can be found in Chapter 6.

There are two major differences between the upgrade of a relational database, and that of an event sourced application: state versus event upgrading and implicit versus explicit schema. Relational databases, usually only save the current state, whether the event sourcing approach is saving all the steps, which gives much more data compared to the event sourcing approach. That is why the efficiency of the upgrade of an event sourced application is considered a important factor in this research. Furthermore, in a relational database the schema is set in the definition of tables and columns, which is not the case in event sourcing.

4.2.2 NoSQL

NoSQL is a data store which is ideal for using a schemaless approach. NoSQL stands for Not Only SQL or often referred to as not relational. As there is no simple definition for what NoSQL is, most scientific literature gives characterizations to NoSQL. The most important ones are (Cattell, 2011; Fowler, 2013a):

- Can scale horizontally
- Is schemaless

- Works well in partitioned/distributed systems
- Is not using the relational database model

These characterizations of the NoSQL data stores are related to the CAP theorem (Section 3.2). They often have or can work with a weaker form of consistency like eventual consistency (Elbushra and Lindström, 2014). This especially has its effect on the scalability. As NoSQL is a set of characterizations, multiple types of data stores are considered being a NoSQL data store. The most important NoSQL data store types are (Cattell, 2011; Sadalage and Fowler, 2012):

Key-Value stores - these systems store values and include a key to find them. Examples of this type of store are Riak and Redis.

Document stores - These store documents. The documents stored are often semi-structured like JSON, XML and so on. Examples of this type of store are CouchDB, MongoDB and RavenDB.

Column-oriented stores - Instead of the standard row-based system, these stores are column based. Examples are Cassandra and Hypertable.

Graph database - The data is stored as a graph, with nodes and the relations between them. Examples of this type of database are Neo4J and OrientDB.

4.2.3 Temporal databases

Because of the similarities between temporal databases and event sourced databases this type of database was also researched. The concept that event sourcing is doing something related to time and order, is not new. Using the time dimension with storage was already done in so-called temporal databases. Clifford (1982) was one of the first who described this type of historical database. Currently, within temporal databases, Three types are defined (Jensen and Snodgrass, 2009). The first one is using time references to save current or past states of the database, which is referred to as transaction time. Within the selected certain time period, the data record was accepted as correct. The second type is referred to as valid time. The main difference between the two is that valid time records what is valid and true in the real world, whether transaction time refers to when we believe or believed the fact in the row. The third type is bitemporal, in which the data has aspects of both transaction time and valid time.

Compared with event sourcing, temporal databases have a little different approach. Temporal databases only save the state, at some point in time, whereas event sourcing is saving the events and not the state. The state at some point in time can be derived. Temporal databases have many similarities with snapshotting.

4.3 Event-driven architecture & event processing

Event-driven architectures are a type of software architecture for applications that detect and respond to events (Chandy, 2009). Key characteristics of such an architecture are: using the publisher/subscriber pattern for events, being asynchronous and being a distributed system (Hohpe, 2006). Event sourcing is a pattern which is considered to be an event-driven architecture with an alternative style for persisting the application state (Erb and Kargl, 2015).

Within the event-driven architectures, there are three general styles how to process the events, focused on analyzing purposes (Michelson, 2006). Often multiple of these styles is used in the same architecture. They are:

Simple event processing - A notable event happens, which initiates an action.
Stream event processing - Both ordinary and notable events happen. Ordinary events are filtered on notability and streamed to information subscribers.
Complex event processing (CEP) - With using complex event processing, you are looking for event patterns, cross event types and (longer periods of) time, which can help in all kind of solutions (Etzion, 2009).

All these event processing types are related to how extensive the analysis of your events is, and what you can do with it. All these processing techniques can be used in combination with event sourcing. Young (2013) presented his approach in which he explained how he made complex event processing possible within his event store framework.

4.3.1 Event transformation

As part of event processing, there exists *event transformation*. Event transformation refers to any operation that takes a single event message or stream of event messages as input and produces a single event message or stream of event messages as the output (Niblett, 2009). The authors identified four classes of transformation types:

Translation - get one event, apply transformation operation and output the translated event.

Split - get one event, return multiple events, all containing a subset of the attributes of the original event.

Aggregation - take one stream of events and use a window (time or size based) to create aggregated event(s) from the events within that window.

Composition - take multiple streams of events, using a window and composition operation to create zero, one or multiple out events.

These transformation types are typical for analyzing purposes, as they make it easier to interpret the events (translation and split) or something summarizes the events so when analyzing you do not have to read all the events (aggregation and composition). Within event sourcing, the events in the event store are the source of truth, the audit trail. You can not simply decide to remove information or to make a summary. These transformation types can happen in some way at the query side in the projections for example in business intelligence or pattern recognition approaches.

Chapter 5

Data transformation operations

Before optimization of the data transformation operations can start, research is needed to know what those operations can be. Therefore the first subquestion of this research is: *Which operations are needed to transform events so that they comply to the new event schema?* This chapter starts with a description of the relevant concepts including their type definitions. After this, all the expected operations are described, including an illustrative example, the type definition, and a visualization.

All the type definitions are done with the Haskell notation[1]. Haskell was chosen because of the strong type system of the language, which helps to identify and illustrate the relation between different operations. For those who are not familiar with Haskell, $\rightarrow$ is the symbol which divides the arguments of the function. The last argument of the type is the type of the result. A function can also be the argument or the result of another function (resulting in higher order functions). To illustrate this, a few examples functions are given:

$exampleFunction :: TypeA \rightarrow TypeB \rightarrow TypeC$

The $exampleFunction$ is getting two input arguments, which result in $TypeC$.

$exampleFunction2 :: (TypeA \rightarrow TypeB \rightarrow TypeC) \rightarrow TypeD$

$exampleFunction2$ is getting a function (the type argument between

[1]Want to read more about Haskell? You can go to https://www.haskell.org/

the parentheses) as input and will result in $TypeD$.

These type definitions are defined as contracts. This means all the (business) logic is left out: only the expected in- and output of the operation are defined. The complete type definition and a simple implementation in the functional language of Haskell can be found on Github[2].

5.1 The structure and schema in the event store

Before the operations are described, a explanation of the structure and the schema of events is given. Next to the textual illustration, the concepts are defined in a more formal way, by a type definition in this section.

5.1.1 Attribute

Attributes are name and value combinations. These can be found in events.
$typeAttribute = (Name, Value)$

5.1.2 Event

Events represent changes to the application state and are raised at the command side of the application. An event consists of a type (or also referred to as the name), timestamp of the approval of the event and the set of attributes which represent what happened and/or changed by the event. The event type is used to find events of interests in the streams. When the events are replayed, the state of the application can be rebuild.
$typeEvent = (EventType, Timestamp, [Attribute])$

5.1.3 Event stream

In an event stream, all the events related to the same aggregate can be found. Within the stream, the ordering of events over time should be guaranteed. This ordering is not guaranteed when combining events from multiple event

[2] https://github.com/AFASResearch/EventSourcing-DataMigration-Types

streams, because of the possibly distributed character of streams. The distributed character gives extra options to scale the application, as each event stream can be stored in a different event store on different machines.
The type ($StreamType$) represents the kind of stream (or also referred to as the name), and the $Source$ is the identifier of the source. Within CQRS, this identifier of the source represents the aggregate root from the aggregate that is responsible for the validation and coordination of the complete aggregate. The $StreamType$ can be seen as the kind of instance, and $Source$ can be seen as a link to the instance. They both exists because there can be multiple instances (aggregates) but $Source$ is unique within the $StreamType$.
$typeStream = (StreamType, Source, [Event])$

A variation can be made in the given types. A command can result in multiple events, but those events represent a single action. That is why some CQRS and event sourcing implementations want to store these events as the same atomic operation, in one transaction. Therefore these events are sometimes bundled in a single commit, which is stored as a single transaction with a set of events combined with one and the same timestamp.
$typeCommit = ([Event], Timestamp)$

$typeStream = (StreamType, Source, [Commit])$

5.1.4 Event store

The event store is the complete set of streams that are used in the application.
$typeStore = [Stream]$

5.1.5 Event schema

Although there might be no explicit schema, the application still needs to know about the structure and type of events: the *event schema*. Part of the schema is:

- supported stream types,
- supported event types,
- which event types occur in which stream types,
- which attribute exists in the event types, and
- what are the types of the attributes.

5.2 The operations

Now all the concepts used including their schema in the event sourced application are defined, identification of the operations which can be expected during the upgrade of an application was started. This was done in brainstorm sessions and later validated in a session with experts from AFAS. The final result can be found in this section. The description of the evaluation session, including the applied changes, can be found in Chapter 9.1.

Operations were identified on multiple levels:

The level of event store - operations related to one or multiple streams.

The level of event stream - operations within one stream, related to one or more events.

The level of events - operations related to one event or event type.

Not only different levels but also different complexity levels were identified. Therefore another categorization was made, now on the basic or complex level:

Basic operations - operations which cannot be expressed by other operations and are not dependent on keeping state during performing the operations.

Complex operations - operations which consist of one or multiple basic operations, which possibly use state in performing the operation. This research describes only the most used ones, which are described to illustrate the options.

From the complex operations, only the most used operations are described, as many different options and unusual configurations depending on your implementation, are possible. The categorization and the complete overview of all the expected operations is visualized in Figure 5.1.

Running example

To illustrate the operations on the different levels, throughout the different descriptions, we will use a running example. Suppose we have an application, in which things are sold to all kinds of organizations and persons. These customers each have a specific aggregate and event stream, where their events are raised. One of the important event types that is raised is

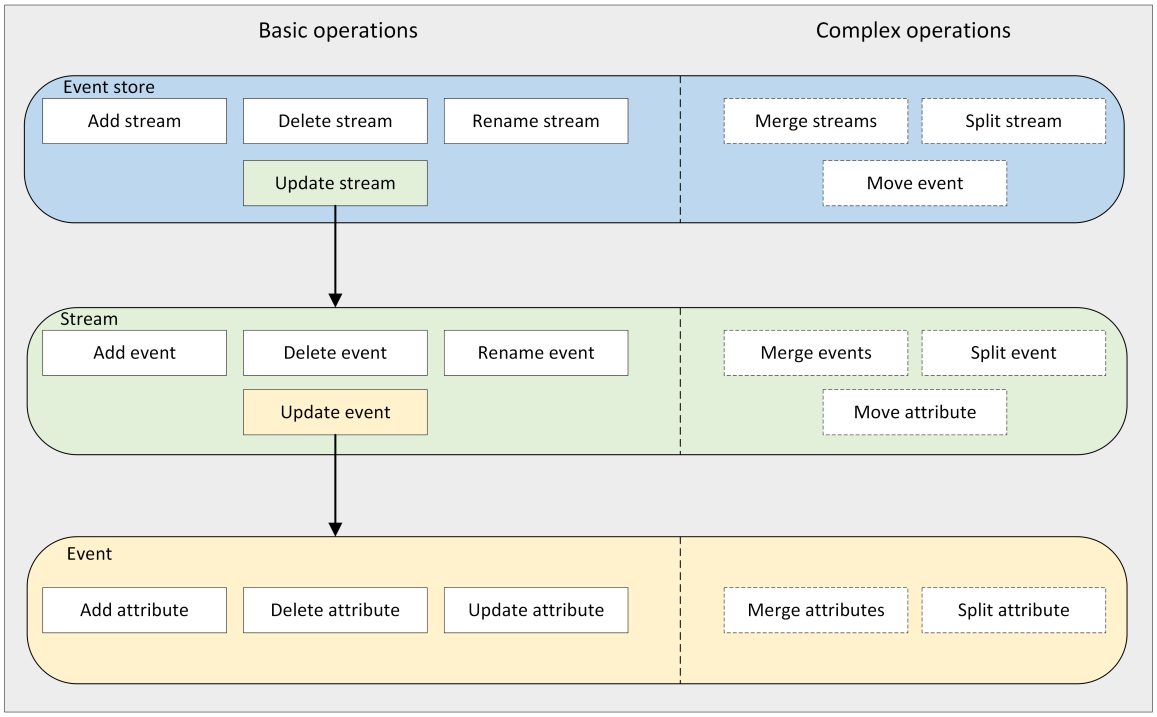

Figure 5.1: Operations overview

$CustomerInformation$, an event type that includes all kinds of information related to them, like name, address and other information which is relevant for them being a customer. In the application, there are also streams found, related to other parts of the business, like persons, employees, and the ordering process.

The most important part of this running example is the current implicit schema. To illustrate how this would look like for this example, the implicit schema is partially defined.

$$StreamSchema = [(Customer, [CustomerInformation, UpdateAddress...], Person, [..]]$$

$$EventSchema = [(CustomerInformation, (Name, Address, Birthday), ...]$$

As the application is upgraded, this implicit data schema is changing. This changing schema is reflected by the operations, which will be described now.

5.2.1 Event level operations

At the lowest level, the operations on the event and the attributes it consists of are expected.

5.2.1.1 Add attribute

This operation adds an attribute within an existing event. The added attribute can either be based on some standard value, or on a condition. For our example, newsletter functionality is added to our application. Therefore the application needs to know if each customer is receiving the newsletter or not. For this reason, an attribute needs to be added during the introduction of the new functionality to the $CustomerInformation$ events, because at that point the knowledge whether a customer needs to receive the newsletter is needed. The functionality is introduced opt-out, so the new attribute is initiated as true. Each customer can now profit from the new functionality. The visualization of this operations can be found in Figure 5.2.

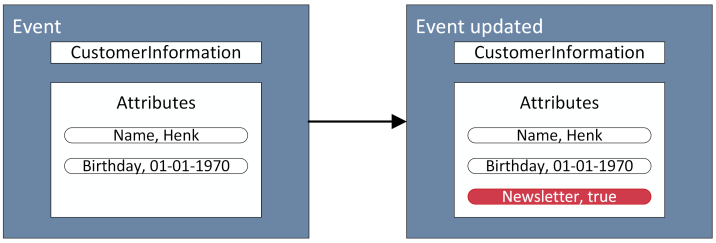

Figure 5.2: Event operation - add attribute

The type of the operation can be seen as a set operation. The operation adds an attribute to the existing set of attributes of the event.

$$addAttribute :: [Attribute] \rightarrow [Attribute]$$

5.2.1.2 Delete attribute

With the delete attribute operations, the attribute needs to be deleted. This can be useful in the case when the attribute is not needed anymore or needs to be deleted because of new privacy, security, or political regulations. In the given domain example, the BSN (the Dutch social security number), of a customer is stored in an $CustomerInformation$ event. Due to new privacy

regulations it needs to be removed from the events, but the rest of the event needs to stay intact (Fig. 5.3).

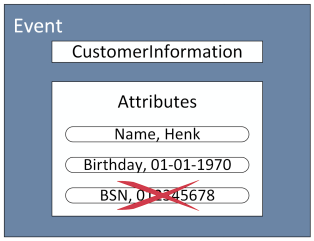

Figure 5.3: Event operation - delete attribute

As a type definition, it has the same type as the "create attribute"-operation. However, in this operation, the attribute is removed from the set of attributes from an event.

$$deleteAttribute :: [Attribute] \rightarrow [Attribute]$$

5.2.1.3 Update attribute

The update attribute operation can be two different operations because both the name and the value can be changed. The name of an attribute from an event is updated. In the example, within the $CustomerInformation$ event, the name of the attribute $BankAccount$ is changed to $BankNumber$ (Fig. 5.4). This operation functions as a rename.

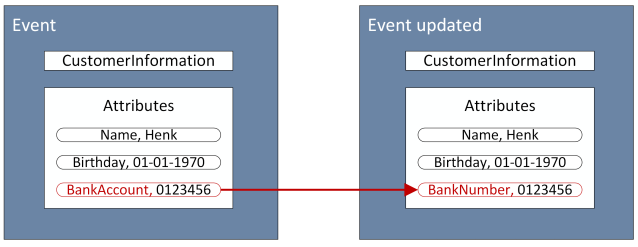

Figure 5.4: Event operation - update name of attribute

The other type of update operation is the update of the value of an attribute. Examples of possible updates are length or type changes, but it can also be functions. Within the context of the example with $CustomerInformation$,

a function was needed when there was an update of the traditional Dutch bank account number to IBAN. The Dutch bank account changed to an international standard in 2014. Therefore the bank account numbers needed an update to this new standard (5.5).

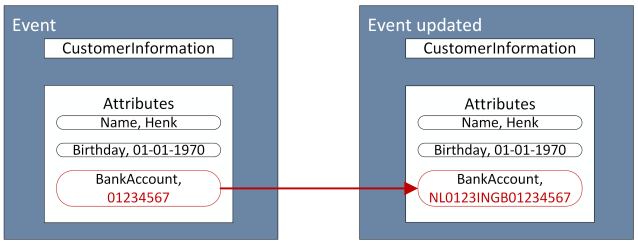

Figure 5.5: Event operation - update value of attribute

The update of an attribute has the same type for both updating the name as the value, from an attribute to an attribute.

$$updateAttributeValue, updateAttributeName :: Attribute \rightarrow Attribute$$

5.2.1.4 Merge attributes

Multiple attributes are merged into one attribute within one event. A good example for this operations is that within our $CustomerInformation$ event. The street and street number are two separate attributes but are only needed in the combination together. Therefore, they can be better be merged to one address attribute (Fig. 5.6).

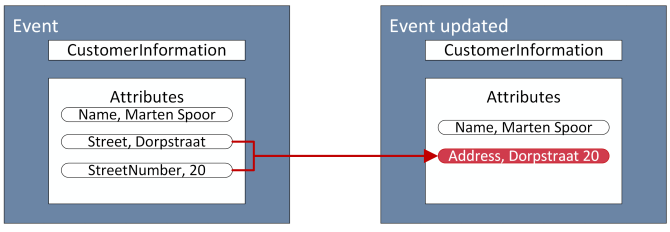

Figure 5.6: Event operation - merge attributes

The merge attributes operation is considered a complex operation. When looking to this operation from a set perspective, again the type would be

from set of attributes to a set of attributes. Therefore, the type is defined as multiple attributes to one attribute.

$$mergeAttributes :: [Attribute] \rightarrow Attribute$$

5.2.1.5 Split attribute

One attribute is split into multiple attributes, all in the same event. Within our example, $CustomerInformation$ consisted only of the name attribute. As there is the wish to use the first name and surname of the customers separately, the attribute needs to be split into two attributes: the first name and surname (Fig. 5.7).

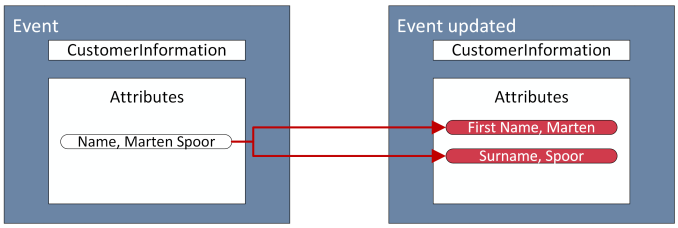

Figure 5.7: Event operation - split attribute

The split attribute operation is considered a complex operation. The type of this operation, being the opposite of merge, is from one attribute to multiple attributes.

$$splitAttribute :: Attribute \rightarrow [Attribute]$$

5.2.2 Event stream level operations

The operations on the level of event stream are operations which have to do with one or more events in the same stream.

5.2.2.1 Add event

Adding an event is another operation that is expected. A good example of this operation is when a specific event is required for functionality after the application upgrade. In the example, a customer is getting an mandatory

API key, that can be used by new features. Therefore, a new event is raised *AddAPIKey* (Fig. 5.8).

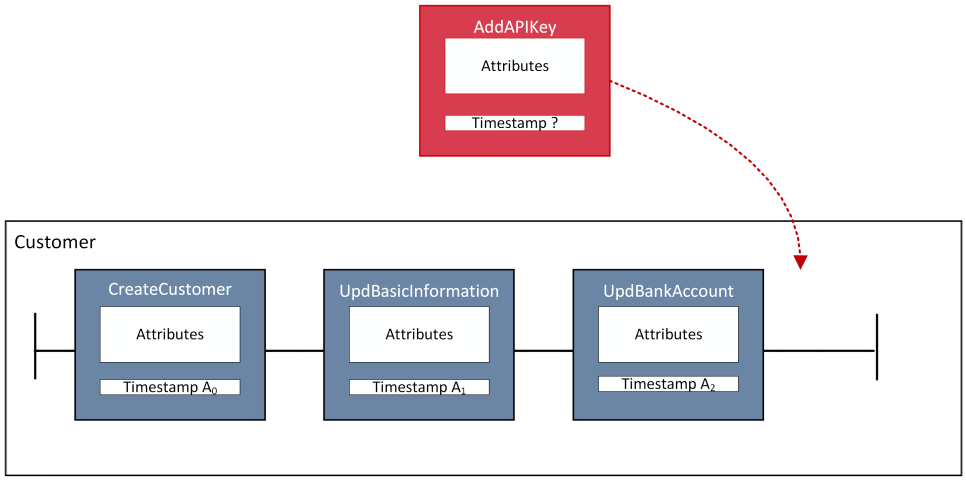

Figure 5.8: Stream operation - add event

The type of this operation is like a set operation, on which there is a set of events and another event is added which results again in a set of events.

$$addEvent :: [Event] \rightarrow [Event]$$

5.2.2.2 Delete event

The delete event operation is the operations which deletes an event from the event stream, for example when there are privacy issues or changing regulations involved. Within the example, credit-card information of the customer was stored in a separate event. This needs to be deleted because of new regulations. Therefore, the entire event needs to be deleted (Fig. 5.9).

The type of this operation can be seen as a set operation on which there is a set of events, and an event is deleted from the set of events.

$$deleteEvent :: [Event] \rightarrow [Event]$$

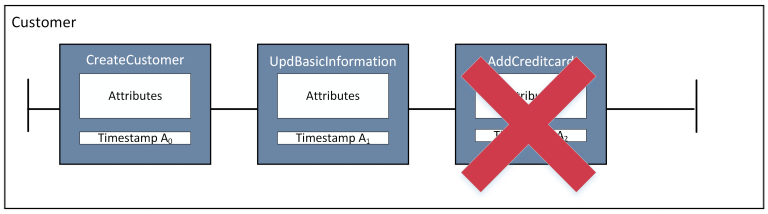

Figure 5.9: Stream operation - delete event

5.2.2.3 Rename event

Renaming the event can be needed, because the original type no longer fits the purpose, or is not correct anymore. In the example, the $CustomerInformation$ event was first called $ClientInformation$, but because of changing requirements terminology was changed, resulting in a rename operation for this event type. Updating the name of an event is referred to as an update to the event type.

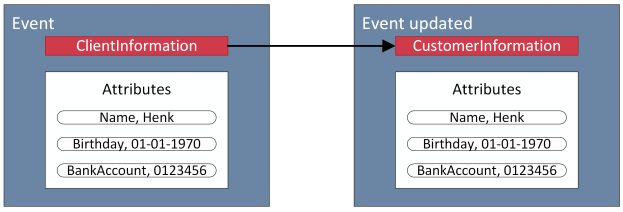

Figure 5.10: Event operation - rename event

The typing of this function is from an $EventType$ to an $EventType$, applied on an event.

$$updateEventType :: (EventType \rightarrow EventType) \rightarrow Event \rightarrow Event$$

5.2.2.4 Update event

Update events is an operation that takes place on the event stream level but consists of operations on the lower level, described in section 5.2.1. Update event consists of one of the operations on the level of events. All the type definitions from the operations on event level fit in $UpdateEvent$, although sometimes a wrapper function is needed.

In the type you find $UpdateEventPredicate$, which defines on what event(type) the operation is performed. Furthermore, the $[Attribute] \rightarrow [Attribute]$ is the operation function, followed by $Event$ which is the event on what the function is applied to.

$$updateEvent :: UpdateEventPredicate \rightarrow ([Attribute] \rightarrow [Attribute]) \rightarrow Event \rightarrow Event$$

5.2.2.5 Merge events

The merge events operation combines multiple events to one, all within the same stream. Within the example, $UpdateAddress$ and $UpdateBankAccount$ were separate events. According to the business, both are never used separately and could be combined in one event, which is $UpdateInformation$ (Fig. 5.11).

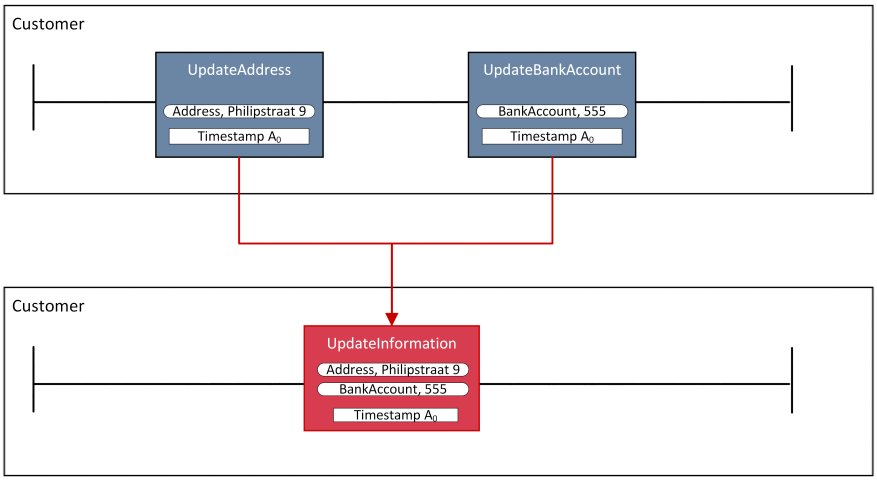

Figure 5.11: Stream operation - merge events

Merge events is a complex operation. As a type definition, this operation is defined from a set of events, to a single event.

$$mergeEvent :: [Event] \rightarrow Event$$

5.2.2.6 Split event

The split event operation splits one event to multiple events, all within the same stream. It functions the other way around as the merge, as can also be seen in the example. Suppose they decided $UpdateAddress$ and $UpdateBankAccount$ could better be separate events instead of combined in the $UpdateInformation$ events (Fig. 5.12).

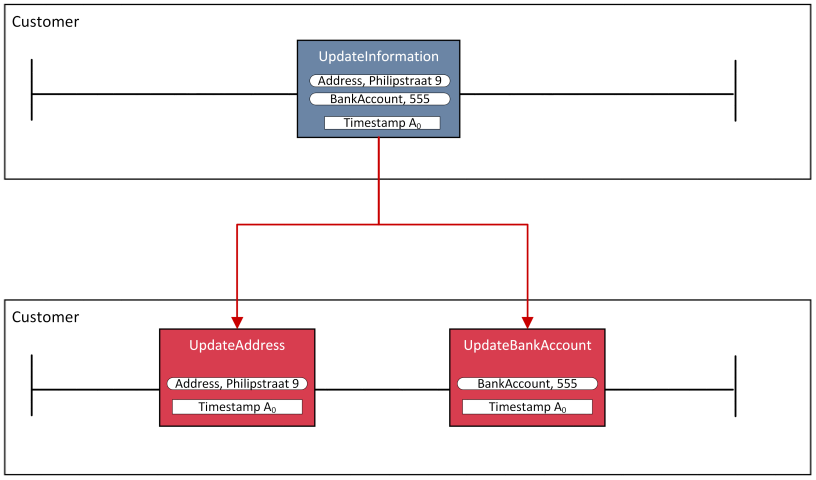

Figure 5.12: Stream operation - split event

The split event is a complex operation. The type of the operation can be defined as from a single event to a set of events.

$$splitEvent :: Event \rightarrow [Event]$$

5.2.2.7 Move attribute

The move attribute operation moves an attribute from one event to another event. Maybe this seems like a strange operation on this level but to move an attribute, two different events are needed. As multiple events are not existing on the level of events, but on the level of the event stream.

Move attribute is a complex operation. The type of the operation can be defined as from a single event, to a single event.

$$moveAttribute :: Event \rightarrow Event \rightarrow Event$$

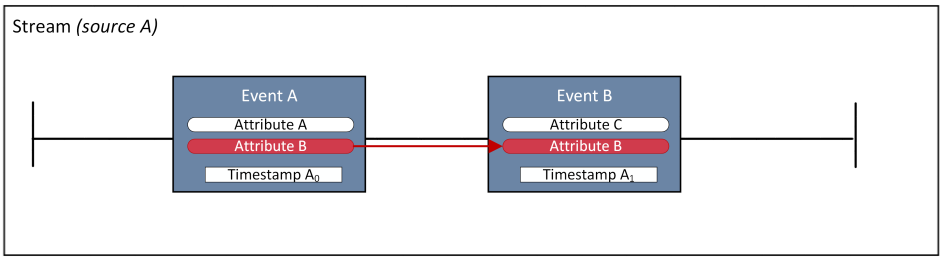

Figure 5.13: Stream operation - move attribute

5.2.3 Event store level operations

We identified operations on the level of the event store. The operations on this level are relevant for one or multiple streams.

5.2.3.1 Add stream

When new functionality is introduced that expects some stream with initiated values to be there, this stream needs to be added during the upgrading process. A good example for this operations is the introduction of (a set of) data tables. In the example from Figure 5.14, the data table from the Netherlands was introduced, which introduces general and necessary information about the Netherlands. This information was not hardcoded into the application to make it easier adjustable and expendable in the future.

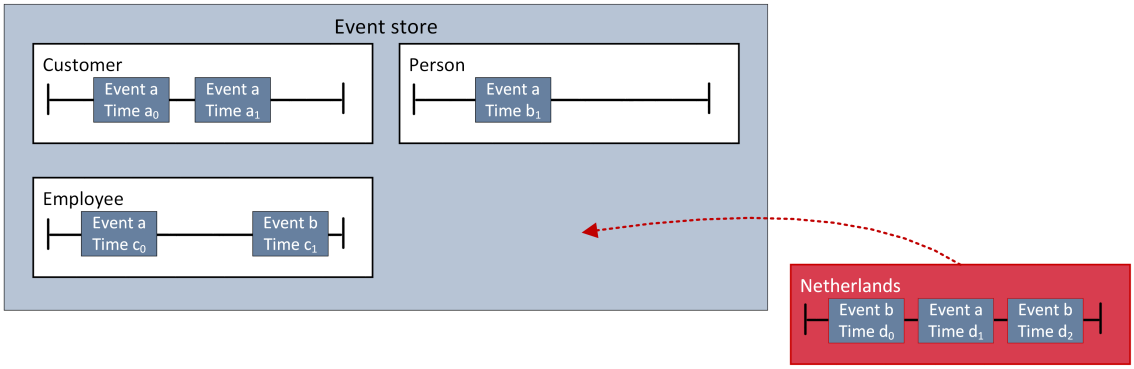

Figure 5.14: Store operation - add stream

The type for add stream is from a set streams to a set of streams.

$$addStream = [Stream] \rightarrow [Stream]$$

5.2.3.2 Delete stream

When some stream becomes redundant, for instance when functionality is removed, privacy or security issues arise, or simply when cleaning up after another operation, the possibility to be able to delete the stream is needed.

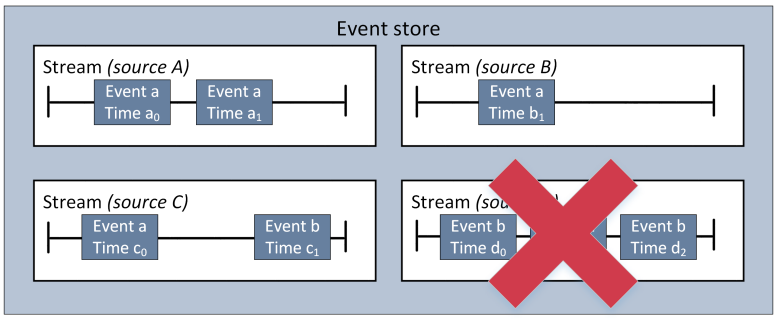

Figure 5.15: Store operation - delete stream

The type for delete is that of a set streams to a set of streams.

$$deleteStream :: [Stream] \rightarrow [Stream]$$

5.2.3.3 Update stream

Update streams is an operation that happens on the event store level, but exists of an operation from the lower level. Update stream consists of an operation on the level of event stream, described in section 5.2.2. All the type definitions from the operations on an event stream fit in *updateStream*, sometimes with the help of a wrapper function.

In the type definition, $UpdateStreamPredicate$ is a selector, for selecting the correct streams on which the operation needs to be applied. This is followed by the function which fits in $([Event] \rightarrow [Event])$ for the stream update and the stream which the update stream operation is applied to.

$$updateStream :: UpdateStreamPredicate \rightarrow ([Event] \rightarrow [Event]) \rightarrow Stream \rightarrow Stream$$

5.2.3.4 Merge streams

With the merge streams operation, multiple streams are combined to one stream. A good motivation for this can be better performance or changes in the functionality of the application. Merge stream usually means that in the application and data scheme, two aggregates are merged and get one and the same stream and aggregate root. Within the example, the streams person and employees are combined (see Fig. 5.16). In the person stream there was only information used in employee records, so combining them gave a performance benefit, since now the connection between the both streams and aggregate roots is not needed anymore.

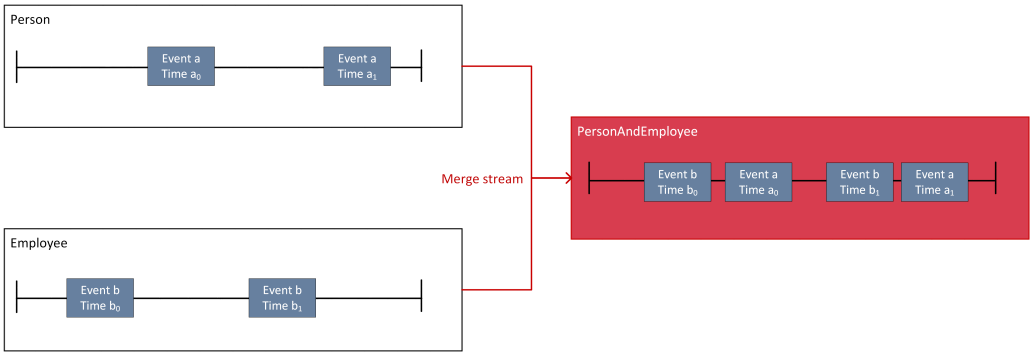

Figure 5.16: Store operation - merge streams

Merge streams is a complex operation. As a type, the operation starts with a set of streams, which are merged to one single stream.

$$mergeStreams :: [Stream] \rightarrow Stream$$

5.2.3.5 Split stream

The split stream is the opposite of merge streams: starting with one stream and create multiple streams from this. Again a possible motivation can be found in performance or changes in the application. Now instead of combining two aggregates, one aggregate is separated in two aggregates and aggregate roots. As an example, the order stream is split. In the current order stream, both the orders and the so-called order lines, lines with order information, could be found. Unfortunately, the order lines became so big, that from the performance point of view the decision was made to separate

them.

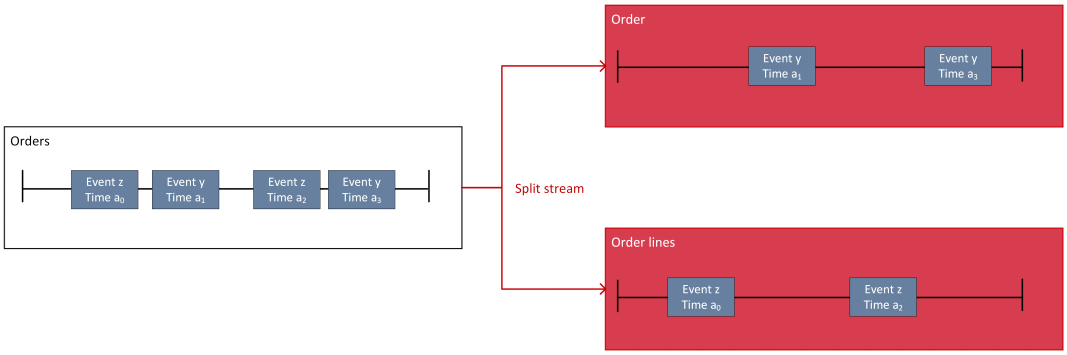

Figure 5.17: Store operation - split stream

The split stream is a complex operation. As a type, the operation starts with a stream, which is split into multiple streams.

$$splitStream :: Stream \rightarrow [Stream]$$

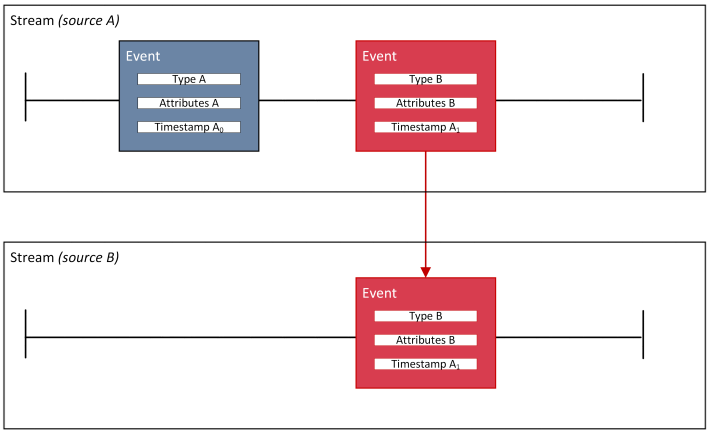

Figure 5.18: Store operation - move event

5.2.3.6 Move event

The move event operation moves an event from one stream to another stream. As this operation needs two streams, which are not there on the level of stream, but on the level of the store, it is considered a store level operation.

Move event is a complex operation. The type of the operation can be defined as from a stream to a stream.

$$moveEvent :: Stream \rightarrow Stream \rightarrow Stream$$

5.2.4 Summary

To give a summary of all the operations, in addition to Figure 5.1, Table 5.1 presents the levels, the operations on these levels, their complexity category and a short description of the operation.

Level	Complexity	Operation	Description
Event	Basic	Add attribute	Attribute is added to an event
		Delete attribute	Attribute is deleted from an event
		Update attribute	Attribute is updated, can be both the name or value(type)
	Complex	Merge attributes	Two attributes are combined to one attribute
		Split attribute	One attribute is split into two attributes
Stream	Basic	Add event	A new event type is added to the stream
		Delete event	An event(type) is deleted from the stream
		Rename event	An event(type) is renamed
		Update event	An event is updated by one or multiple event operation(s)
	Complex	Merge events	Multiple events are combined to one
		Split event	One event is split into multiple
		Move attribute	One attribute is moved from one event type to another
Store	Basic	Add stream	A new stream is added to the store
		Delete stream	A stream is deleted from the stream
		Rename stream	A stream is renamed
	Complex	Merge streams	Multiple streams are combined to one stream
		Split stream	One stream is split into multiple streams
		Move event	An event is moved from one stream to another stream

Table 5.1: All the operations including a description

Chapter 6

Techniques for data transformation

Now that the operations are identified, techniques are needed to execute those operations on the data in the event store. This chapter presents an overview of techniques that can be used to apply data transformation operations.

As part of the scope, the output of the data transformation needs to be complete. All the information including the history needs to be there after deploying/applying the technique. The techniques like converting snapshots or other techniques that are losing history information, are not discussed in this chapter.

In the first section (6.1) of this chapter, schema evolution and versioning is described. Schema evolution and versioning are the angles used to categorize and compare different techniques. After this, in section 6.2, different techniques from both relational databases and then schemaless data(base) structures are discussed. Then, techniques for data transformation in an event sourced application are described, including their advantages and disadvantages (section 6.3). In the final section, a comparison between the different event store techniques is made.

6.1 Schema evolution versus versioning

The terms *schema evolution* and *schema versioning* play a crucial role in data transformation. Roddick (1995, 2009) defined these terms as:

Schema versioning *is accommodated when a database system allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces.*

Schema evolution *is accommodated when a database system facilitates the modification of the database schema without loss of existing data.*

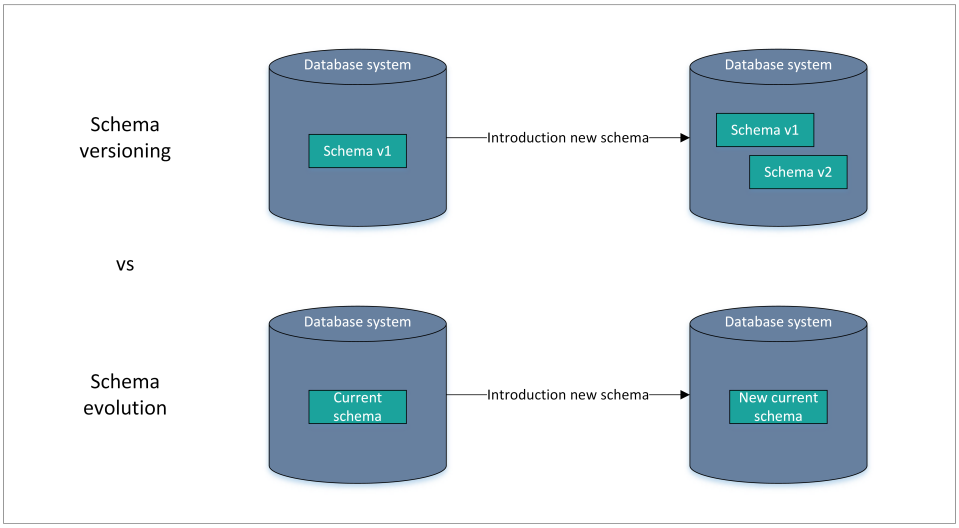

Figure 6.1: Schema versioning vs. evolution

From the definitions and illustration in Figure 6.1, you see that in schema versioning data is accessible through multiple schema versions at the same time, compared with schema evolution where the data is only accessible through one schema version. These characterizations of being a schema evolution or schema versioning technique, are found on data transformation techniques throughout the different data stores.

Figure 6.2 shows how schema versioning and evolution relate to the architecture and data transformation. It illustrates that schema versioning techniques are (mostly) related to elements outside the database, like the application, or some layer in between. The database needs to be able to handle multiple schemas at the same time, while the data itself is not changing. Examples for a layer between the application and the database is the serialization or deserialization of data. This is rather logical because although the data is not changed within the database. Since multiple schemas exist next to each other, the data from multiple schema versions logically needs to be combined. As the figure illustrates, schema evolution techniques are related to the database. This follows from the fact that the database system

only has the most recent schema and when a newer version of the schema is introduced, the old data needs to be updated accordingly.

There is one important sidenote to the definition of schema versioning. Two types of schema versioning exist: partial schema versioning and full schema versioning. Partial schema versioning is being allowed to view data from all the different schema's, but only data updates through one schema. Full schema versioning allows both views and data updates from all the different schema versions. Within the context of event sourcing, the schema versioning is considered to be partial schema versioning, with the one data update schema being the current one.

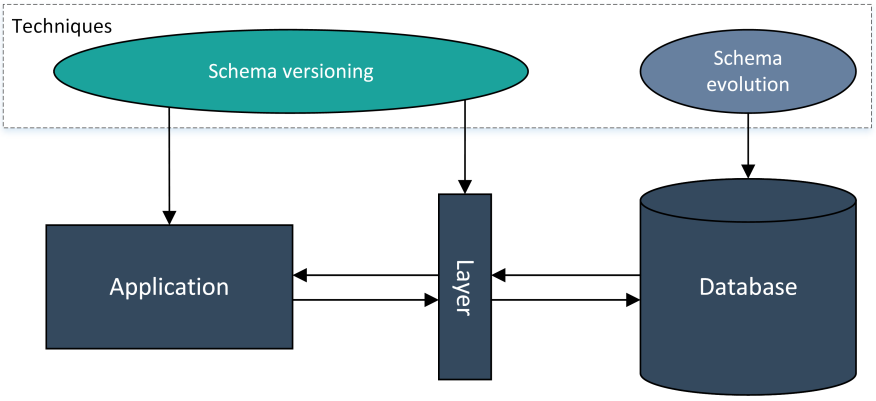

Figure 6.2: Illustration to show where schema versioning and schema evolution techniques live in the system

6.2 Techniques for other kinds of databases

In this section interesting techniques, approaches, and tools. These are categorized in a relational database or schemaless approach. Furthermore, they are categorized as being schema versioning or evolution.

6.2.1 Relational database

Relational databases have been around for some time. Therefore many different techniques and supporting tools exist. The most important ones that were found during our literature research are described in this subsection.

6.2.1.1 Schema versioning

The first solutions of schema versioning on a relational database are already some years old. According to Roddick (2009), amongst the first solutions were GEMSTONE (Penney and Stein, 1987) and ENCORE (Zdonik, 1986). However within the field of schema modifications, the most attention goes to schema evolution, and what to do with the data.

Recently schema versioning is discussed in tools that describe how to version your database, for example like the description by Humble and Farley (2010). The approaches you see are creating backward and forward compatibility migration scripts. This approach is used by tools like SchemaSync or DbDeploy(.NET).

6.2.1.2 Schema evolution

As the starting points when searching for interesting techniques, Hartung et al. (2011) and de Jong (2015) were used. Most of the schema evolution approaches are reflected by tools and experiments. Apparently, these approaches usually are a combination of a technique and applying a deployment strategy. Therefore you will find a description of the most interesting approaches and tools:

Facebook - Online Schema Change - Facebook released a tool they created (Callaghan, 2010). It exists of 4 phases: copy the original database, change the copy to the new schema, replay the changes on the original database that happened during copy/build phase and the switch phase.

IMAGO - IMAGO upgrades the system using what they call a parallel universe, to reduce upgrade failures. This approach was a result of research by Dumitras and Narasimhan (2009a,b). The parallel universe is a system that runs parallel to the system that is upgraded. Due this approach IMAGO can isolate the production system from the upgrade operations and completes the upgrade as an atomic operation.

MeDEA - MeDEA is a generic architecture evolution tool, that focuses on traceability of artifacts, which is based on the research by Hick and Hainaut (2006) and Domínguez et al. (2008). MeDEA makes it possible to translate changes to the conceptual model of a relational database, to schema changes in the actual database.

OpenArk Kit/ Percona Toolkit - OpenArk kit (Noach, 2014) and Per-

cona Toolkit (Nichter and Baron, 2016) are two very similar tools, both using an expand and contract approach (see chapter 7.1). The main difference is that Percona has (some) support for foreign keys, which is not available in OpenArk Kit.

PRISM - PRISM is a (GUI) tool, which can be used by a Database Administrator to perform schema evolution. It is based on research by Curino et al. (2008) and later evolved in PRISM++ (Curino et al., 2013). PRISM can be used to calculate which SQL operations need to be performed for the new schema. It helps to rewrite queries and can check the information preservation, backward capabilities, and the redundancy.

QuantumDB - QuantumDB is a tool created as the result of the research by de Jong (2015) & de Jong and van Deursen (2015). It integrates the deployment strategy expand contract/blue-green (see chapter 7.1), with a schema evolution tool.

When researching these schema evolution approaches, there is much overlap. Most of them use some parallel approach as a deployment strategy. By creating a duplicate database or ghost tables, the technique that is applied is often not blocking the running application, meaning it can be performed with zero downtime. As a technique, they either transform the duplicated database in place, with some catching up phase when making the switch to the upgraded application, or they replay the records from the original tables to the created ghost tables and transform them during the replay. None of these approaches can be considered runtime techniques.

6.2.2 Schemaless data(base)

Like relational data transformation techniques, there are also techniques for "schemaless" approaches. According to Sadalage and Fowler (2012), schemaless databases can use the same migration techniques as databases with strong schemas because they have an implicit schema.

6.2.2.1 NoSQL

Scherzinger et al. (2013) describe how to manage schema evolution in NoSQL data stores. They defined evolution operations and a schema evolution language. They use the eager approach (just run it in a batch), but also describe the option of a lazy approach. According to authors, ideally, the eager approach is better for bigger changesets and the lazy approaches for minor

changes. Sadalage and Fowler (2012) describe a form of a lazy approach for NoSQL, which they refer to as, incremental migration. They describe the lazy approach as an approach that migrates the data to the new schema when it is accessed.

6.3 Event store techniques

There are several techniques for schema upgrading in an event sourcing application. The techniques will be described in this section. The query side is kept out of scope are for these techniques as the focus of this research is on the event store.

6.3.1 Support multiple versions

The *support multiple versions* approach is to support multiple versions of events supported in whole the application. This means that all components that need events, like the aggregates and projections, need knowledge of the different event versions and how to handle them. This approach keeps the event store immutable. When adding new events, they will be according to the newest event schema. This approach is for example suggested by Betts et al. (2013).

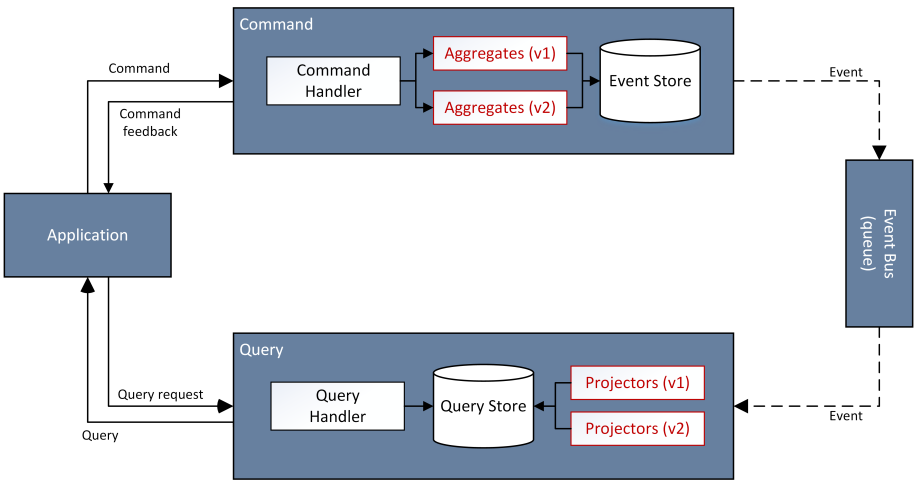

Figure 6.3: Support multiple versions

A big advantage of this approach is that the technique keeps the event store untouched, which stays immutable. One of the main downsides with

this approach is that you create much extra maintenance, with all the extra code to handle the different event versions on multiple places.

One other important issue is that operations that are related to multiple streams, are needed for some of the event store operations. These operations will probably take too much time to be realistically performed in a runtime solution. The big advantage is that *support multiple versions* is a runtime technique that only needs to be deployed, after which it will work directly.

Advantages and disadvantages

+ Application can start using the new schema directly after deployment (runtime technique)
+ The event store is immutable
− Maintenance of this solution is high because each element which uses events (aggregate, projector) gets x times the different event versions
− Not all the operations are supported at runtime

6.3.2 Upcasting

With *upcasting*, the idea is to support multiple versions of a specific event. However now there is only one point in the application that handles events of multiple schema versions and combines them. When comparing *upcasting* with *support multiple versions*, the difference is that that instead of doing *upcasting* around the complete application, it only needs to be applied in one place in your application. The upcasters between event versions can also be chained, which makes it easier to add new upcasters and to do upcasting across more than one event version. Again, this approach leaves the event store untouched. This solution is for example suggested by Axon Framework (2016); Betts et al. (2013).

Upcasting is a useful technique when the changes are small, so this can be integrated with reading the events from the event store. When storing your events in a serialized matter (like JSON/XML), this can be done during the deserialization step, although this is not true for all operations. When many upcasters are chained, the performance of the application is lowered, as it needs to do many transformations of the events.

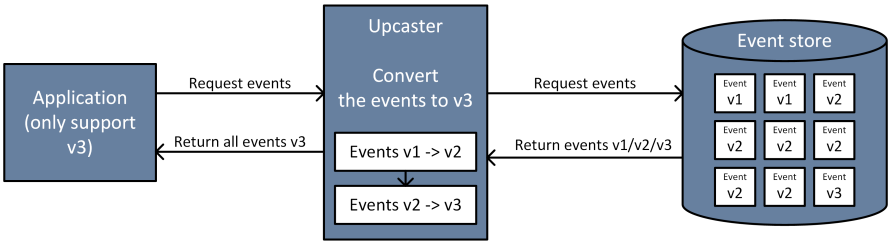

Figure 6.4: Upcasting

Advantages and disadvantages

+ Application can start using the new schema directly after deployment (runtime technique)
+ Only one place with multiple event versions support (compared to *support multiple versions*)
+ The event store is immutable
– Many (chained) upcasters has a negative effect on the performance of the system
– Not all the operations are supported at runtime

6.3.3 Lazy transformation

Lazy transformation is similar to *upcasting*, with one big difference: after the events are upcasted to the latest version, the upcasted events are used to update those specific events in the event store. This means the event store does not stay untouched but updates the events in the event store. This "lazy technique" is similar to techniques which were found in other transformation techniques, like the incremental approach (Sadalage and Fowler, 2012) or in relational approaches (Roddick, 1995; Tan and Katayama, 1989).

The major difference between *lazy transformation* and the previously suggested techniques is that the event store is not immutable anymore. An advantage of this technique is that it means that all the events are only upcasted once because after the upcasting the newest version is updated in the event store and upcasting is not needed anymore. One of the bigger downsides of this technique is that you do not know when the lazy part is done, as you do not keep track of the upgrading process.

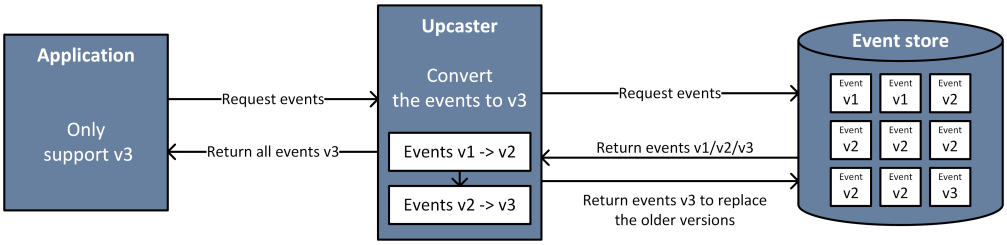

Figure 6.5: Lazy transformation

Advantages and disadvantages

+ Application can start using the new schema directly after deployment (runtime technique)
+ Only one place with multiple event versions support (compared to *support multiple versions*)
+ Old events are requested and upcasted only once
– Many (chained) upcasters has a negative effect on the performance of the system
– Not all the operations are supported at runtime
– This technique is unpredictable because it is unknown when it is done
– The event store is not immutable anymore

6.3.4 In place event store transformation

In place event store transformation is in essence a simple technique. The main idea behind the technique is to run a script that transforms the current event store to the updated schema. The major advantage of this approach

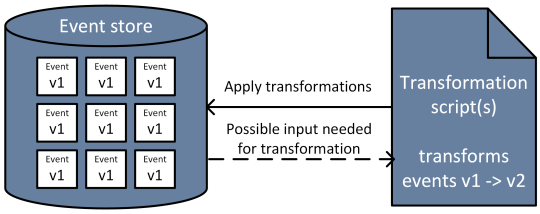

Figure 6.6: In place event store transformation

is that all the operations are supported because it is not called when accessing your data for usage in your application, but outside by this script. Furthermore, within the architecture, only the event store is affected, while the other components are not. A downside is that it will possibly lock up (parts) of your event store when it is transforming it. Another downside is that as it needs to transform all the changed events, it will take some time to be completed.

Advantages and disadvantages

+ All the operations are supported
+ The transformation does not have any impact on the rest of the system
− This approach will take much time
− Can put a lock on (parts) of the event store when it is running
− The event store is not immutable anymore

6.3.5 Replay the event store

With the *replay event store* technique, you are going rebuild the event store from scratch. This is done by replaying all the events from the current event store to an empty event store. When replaying the events to the empty event store, the events that need transformation, are transformed to the updated schema.

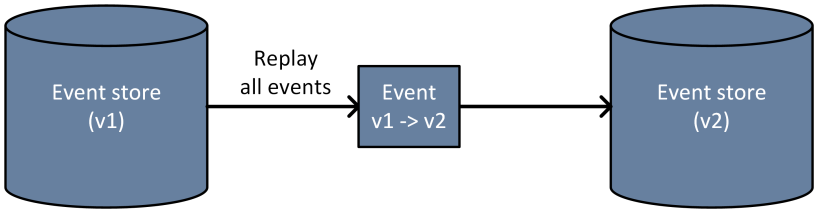

Figure 6.7: Replaying

Replay the event store is a technique that can be used to execute any operation as it is not done runtime. Another big advantage is that it gives the option for a rollback strategy because the current event store stays intact. The big downside is that this is the slowest approach of all the suggested

techniques because all the events need to be transported to the clean event store.

Advantages and disadvantages

+ All the operations are supported
+ The transformation does not have any impact on the rest of the system
− This approach will take much time
− The event store is not immutable anymore

6.4 Analysis and comparison of different event store techniques

To be able to analyze and compare the different techniques, the techniques are mapped to figure 6.1. This to see the relation between schema evolution and versioning, whether they are runtime or not, and where the technique is placed in the spectrum of application, layer or event store. Furthermore, all the techniques are compared according to a set of quality attributes.

Multiple versions and *upcasting* are mapped to schema versioning because these techniques keep multiple schema versions active in the event store. *In place event store transformation* and *replay the event store* are mapped to event schema evolution, as they both are about keeping one schema active and updating the data accordingly. *Lazy transformation* is the one that stands out since it is a solution which is runtime, but at the same time evolving the schema. Overall it is considered a schema evolution technique as the end result of the technique is an event store according to only one schema. This mapping can be found in Figure 6.8.

To compare the different techniques, relevant quality attributes of the ISO/IEC 25010:2011 are used (ISO/IEC, 2011). As they are general quality attributes, for each of the quality attributes both the definition and the instantiation, which describes how they are relevant to the techniques, are described. These relevant quality attributes are functional completeness, maintainability, performance efficiency, and recoverability.

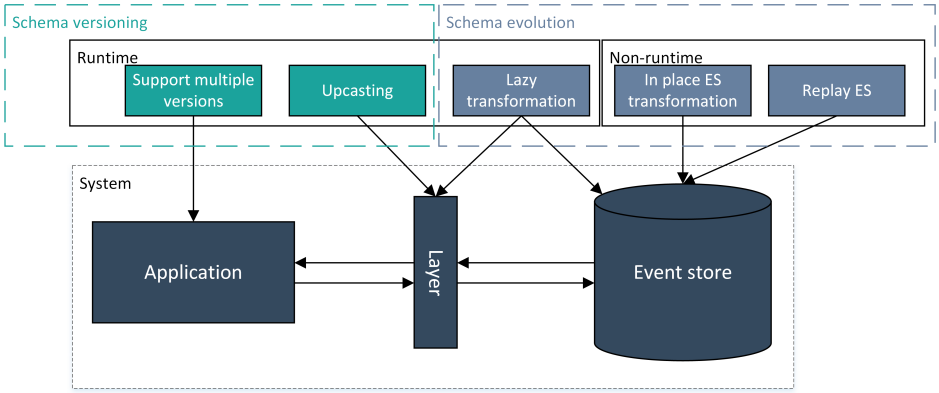

Figure 6.8: Techniques initiated

Functional completeness - is defined as the degree of which the set of functions covers all the specified tasks and user objectives. Within our comparison, it instantiates as: Can the technique perform all the operations?

Maintainability - is defined as the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. Within our data transformation technique, it instantiates to how easy it is to apply the technique and how big of a factor is it on the existing architecture.

Performance efficiency - is defined as performance relative to the amount of resources used under stated conditions. Instantiated for this problem as the duration for the technique to perform the operations. How is the performance runtime? How is the performance deployment time?

Recoverability - is defined as the degree to which, in the event of an interruption or a failure, the product or system can recover the data directly affected and re-establish the desired state of the system. Instantiated to this problem, this quality attribute is related to what is the effect when something in/during the technique goes wrong?

6.4.1 Applying the quality attributes

Now that the quality attributes are selected, the techniques can be compared using these quality attributes. The comparison is depicted in Table 6.1. The results will be discussed per quality attribute.

Functional completeness - For this quality attribute the techniques ended up in two groups: the group which is doing the operation at runtime (*multiple*

versions, upcasting and *lazy transformation*) and the group of techniques which are transforming the event store (*in place event store transformation and replay event store*). As it is expected that the operations across multiple aggregates are too complex and therefore too slow for a runtime solution, the runtime techniques are characterized as less functional complete.

Maintainability - *In place event store transformation* and *replay event store* both scored + for maintainability, because after these techniques are finished, you have no maintenance anymore regarding the update of the schema. *Support multiple versions* scores the lowest because you have the upcasting mechanism on multiple places for each version.

Performance efficiency - *Multiple versions* and *upcasting* are considered performance efficient, because when these techniques are deployed in the new application, it works directly. Of course, this is also the case for *lazy transformation*, but as *lazy transformation* writes back to the event store, this can decrease the performance of the system. As *In place event store transformation* and *replay the event store* are not runtime solutions, these will be judged mostly on deployment time. During deployment *In place event store transformation* touches only the events that are needed for the transformation which is the reason why it scores ±. *Replay the event store* is touching all the events because they all needed to be transported to the empty event store. As this costs much time, it is seen as the most performance inefficient.

Recoverability - As both *multiple versions* and *upcasting* do not touch the original event store, recoverability is high. *Lazy transformation* is dependent on when the recoverability needs to be done. It can be the case that some events are already upcasted and the old events are overwritten. *In place event store transformation* scores bad in recoverability, as it is makes its changes in the original and only existing event store. *Replaying* is scoring good at recoverability because it only replays the original event and does not make changes to the original event store. To recover the event store in case of trouble you can just return to this original event store.

Quality attribute	Support multiple versions	Upcasting	Lazy transformation	In place event store transformation	Replay event store
Functional completeness	±	±	±	+	+
Maintainability	-	±	±	+	+
Performance efficiency	+	+	±	±	-
Recoverability	+	+	±	-	+

Table 6.1: Comparison of event store techniques according to four different quality attributes. The + means this technique is scoring good on the quality attribute, ± means could be better, but not bad. − means it scores bad on the quality attribute.

Chapter 7

Deployment strategies

Now the techniques are identified, knowledge is needed on how these techniques can be executed. Therefore a literature study was performed to finding the deployment strategies. After the strategies are found, they are analyzed in the final section.

7.1 Strategies

According to Humble and Farley (2010) deploying software involves three phases:

Prepare and manage the environment in which your application will run and be deployed. This has to do with infrastructure-related aspects, like the hardware, but also the software that is running on it.

Installing the correct version of what you wish to deploy on it.

Configuring your application, including the data/state that is needed.

The interest of this research lies in the second and third point. How to get your application deployed and your data/state in line with the newly deployed application? The installing phase is called *application upgrade* in this research. Configuring is called *data upgrade* in this research.

For these deployment strategies, the focus was on strategies purely related to the deployment of an application. This excludes strategies like feature flagging, dark launching, and canary release. These deployment strategies are variants of other strategies, used to gain more knowledge about the

users and/or (system) performance.

Five different strategies were identified, which were relevant within our scope. For all the five strategies a description is made, including an illustrative figure and a small description of the advantages and disadvantages. This chapter ends with analyzing the deployment strategies in the light of being a *data upgrade* and/or an *application upgrade* strategy.

7.1.1 Deploying on running system

First the simplest deployment strategy. Just put the new application in place, on your server. Brewer (2001) refers to this approach as fast reboot. The only downtime you have with this strategy is the deployment time. Just reboot the system with the new application deployed. The main advantage is that this strategy is not very complex.

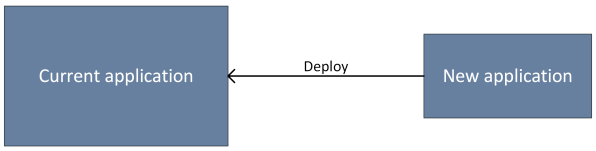

Figure 7.1: Deploying on running system

7.1.2 Big flip

Big flip is a strategy in which the load balancer plays an important role. The server park is divided into two groups. The load balancer routes everyone to one group, but at that time, the system is upgrading/deploying the new application to the other group. When the deployment is finished at that set, the load balancer routes all the users to the upgraded group. When this is done, the other group can be upgraded. After this is done, all the servers are upgraded, and the load balancer can send the users to all the servers again (Brewer, 2001). The biggest downside of this approach is that the amount of active servers that is running the application is reduced by half. The advantage is that the users should not have any downtime.

Step 1: The current situation. A load balancer is in place, and there is a cluster of servers.

Step 2: The servers are divided into two groups. One is upgraded to the new version, and the load balancer redirects everyone to the group that is not upgrading.

Step 3: After the upgrading group is finished, the load balancer redirects all users to the already upgraded group and the other group is turned inactive so that group can now be upgraded.

Step 4: As the second group is also done upgrading, the whole cluster is upgraded and *big flip* is completed.

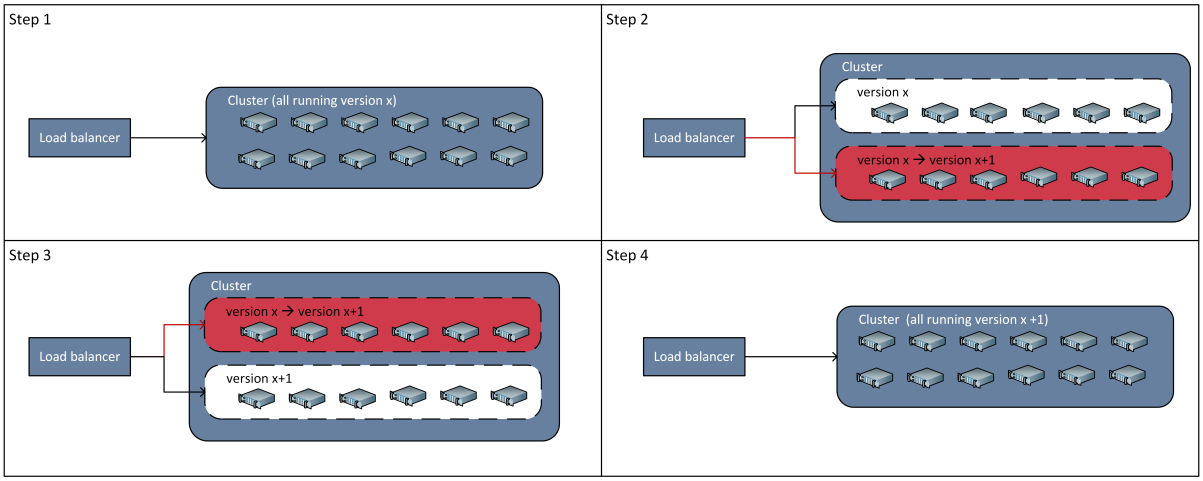

Figure 7.2: Big flip

7.1.3 Rolling upgrade

Rolling upgrade is very similar to *big flip*. The only difference is that the cluster is not divided into two sets, but rather they are upgraded in waves. This means more than two sets. The big difference between *rolling upgrade* and *big flip* is that *big flip* switches the complete system between version and version + 1. With *rolling upgrade* you can have both versions active at the same time, which adds extra complexity. The upside compared to *big flip* is that there are fewer servers that are not running. In Figure 7.3, the *rolling upgrade* is illustrated. Step by step it works as follows:

Step 1: The current situation. A load balancer is in place, and there is a cluster of servers.

Step 2: The servers are divided into more than two sets. One set is upgraded

to the new version, and the load balancer redirects everyone to the sets that are not being upgraded.

Step 3: Upgrading of the first set is finished, and these servers can be used again. Another set is starting to upgrade. The load balancer can redirect users to the already upgraded group or to the group that is still at the current version and is not being upgraded yet. The load balancer can currently redirect the users to two different active versions.

Step 4 - Final step-1: Step 3 is repeated until the last set of servers is being upgraded.

Final step: As the final group is also done upgrading, the whole cluster is upgraded and the *rolling upgrade* is completed.

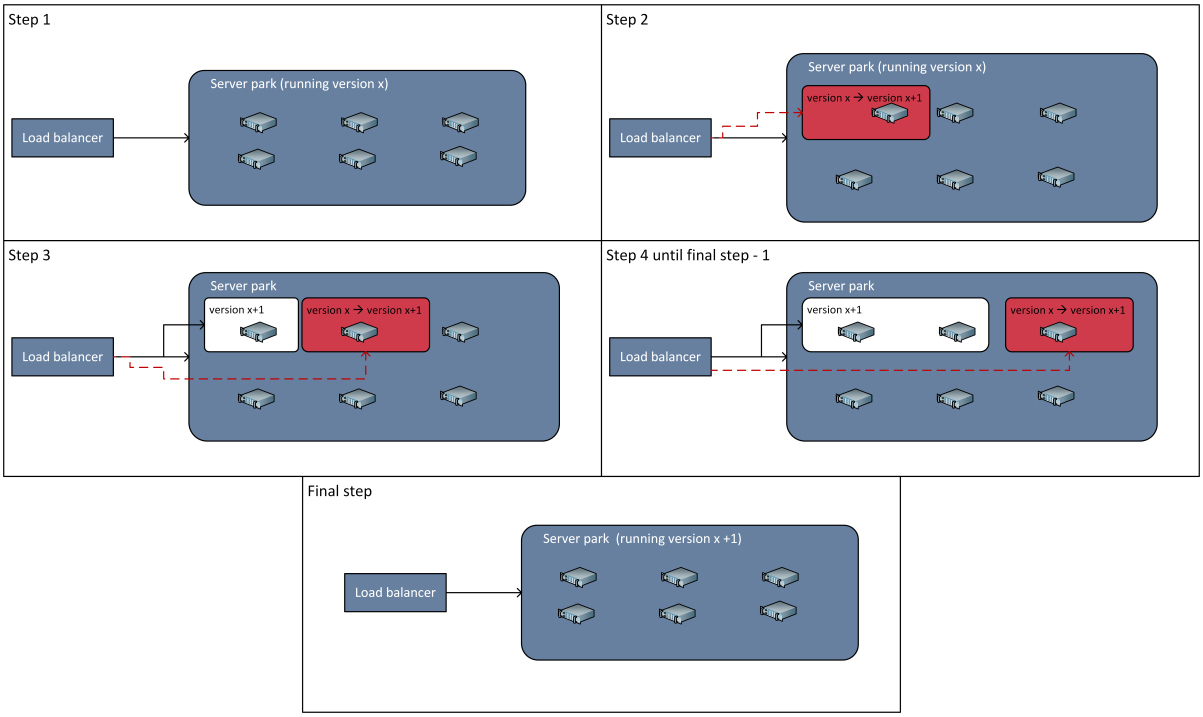

Figure 7.3: Rolling upgrade

7.1.4 Expand-Contract DB deployment

This deployment strategy is an implementation of the "parallel change"-pattern. This "parallel change"-pattern, or "expand and contract" (Sato, 2014), is happening in three phases:

- Expand phase - An interface is created to support both the old and the new version
- Migrate phase - the old version(s) are (incrementally) updated to the new version
- Contract phase - the interface is changed, so it only supports the new phase

This strategy is used in QuantumDB, Percona/OAK, and Facebook's online schema change. In the database, ghost tables are created for the new version of the schema, kind of like a second database.

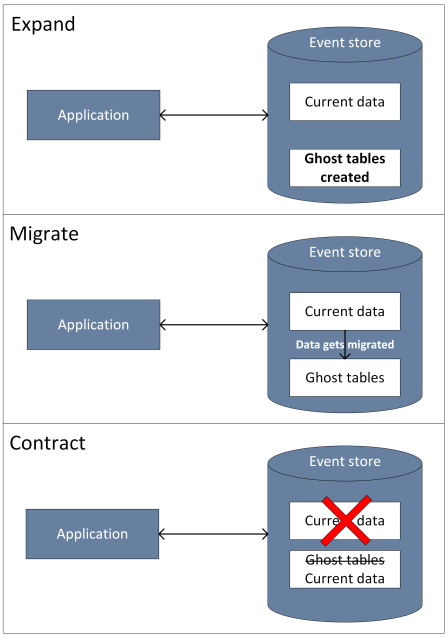

Figure 7.4: Expand-Contract

This is shown in figure 7.4. To explain the strategy a bit better, an example:

Expand: *Persons* is the original table. *Persons_v2* is created as a ghost table, according to the new schema.

Migrate: The original records from *persons* are migrated to the new *persons_v2*, in a non-locking way. Currently, changes happening to *persons* are directly triggered and synchronized in *persons_v2*. When this is not possible because of lock issues, these are logged and migrated to *persons_v2* when the

application is not active, or during the application upgrade in the contract phase.

Contract: When the migrate phase is done, the application needs to be upgraded from *persons* to *persons_v2*. During that upgrade *persons* is removed and *persons_v2* is renamed to *persons*. After this the deployment is finished.

One of the advantages is that the original data is updated to the final schema, without locking the event store, because it is done in parallel. As a downside, this solution has high complexity because operations do not need to lock for the rest of your system. Furthermore, you want to trigger ongoing changes to the application or store them somewhere.

7.1.5 Blue-green deployment

The *blue-green deployment* strategy is one of the most powerful techniques for managing releases according to Humble and Farley (2010). This approach also uses the "parallel change"-pattern, but on the level of servers instead of creating two slots in the data store.

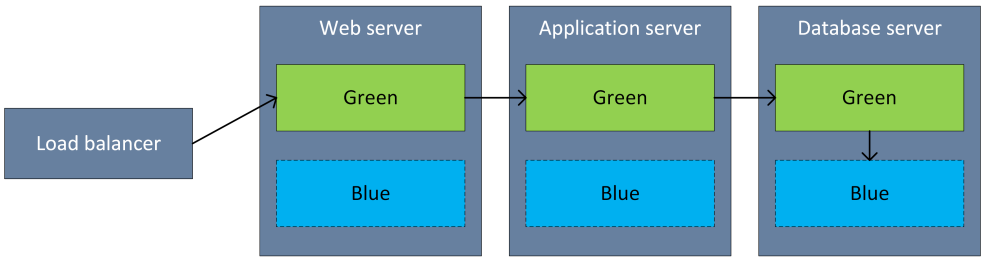

Figure 7.5: Blue-green deployment

The green environment are the instances that are currently in use. The new application is deployed to the newly created blue environment. This deployment includes a data transformation when needed. When the complete deployment including data transformation is done, you let the router switch from green to blue and it is finished (Fowler, 2010).

This strategy has the major benefit that it is a good rollback strategy since the original environment is always alive. Furthermore, the strategy is not locking on the active event store. One of the downsides is that it is a time-consuming solution, which also requires extra resources, because of complete duplication.

7.2 Deployment strategies analysis

As the relevant deployment strategies are identified, we need to know from each strategy whether it would fit as an application upgrade strategy, a data upgrade strategy, or both.

Strategy	Application upgrade	Data upgrade
Deploy on running system	✓	✓
Big flip	✓	✓
Rolling upgrade	✓	✓
Expand-contract		✓
Blue-green	✓	✓

Table 7.1: Strategies application/data deployment

Deploy on running system - can be done both for the application upgrade and for the data upgrade. The only problem is that it will always be without zero downtime.

Big flip is a strategy that is mainly focused on the application upgrade but is also applicable as a data upgrade strategy. For using it as a data upgrade strategy, the database servers need to be aligned with the application servers, to create two pools. When using this on the database, this will lead to downtime because of the data upgrade aspect. This because you do not have any parallel or duplicate version running. A side effect of using *big flip* is that on whatever it is applied, you will have less active servers to handle the requests.

Rolling upgrade is a strategy that is similar to *big flip*, again for the application and data upgrade part. It has one slight difference since the load balancer needs to know which customers started working on the newest version. When customers started working on the new version, the load balancer cannot send them back to the old version. A side effect of using a *rolling upgrade* is that you will have less active servers to handle the requests.

Expand-contract DB is only a data upgrade strategy since it involves migrating data, which is not needed for the application upgrade part.

Blue-green deployment can be both an application upgrade strategy as well as a data upgrade strategy. As an application upgrade strategy it quite

similar to *big flip*, only having a rollback strategy, as the other side is not cleared. Because you do not half the active servers when upgrading, as the application always have one active group and one not active with blue green. This also works as a data upgrade strategy.

This resulted in the following overview, which can be found in Table 7.1.

Chapter 8

The design of an upgrade strategy

To execute the operations efficiently, the operations, techniques, and strategies need to be related to each other. First, the operations and techniques will be related. Then the techniques need to be deployed with a deployment strategy. In describing the relations, the focus will be the efficiency aspect, which means both zero downtime, and not taking too much time. When all the relations are described, the framework is completed.

8.1 Operations and techniques

Operation level	Support multiple versions	Upcasting	Lazy transformation	In place event store transformation	Replay event store
Event - basic & complex	✓	✓	✓	✓	✓
Stream - basic & complex	✓	✓	✓	✓	✓
Event store - basic	✓	✓	✓	✓	✓
Event store - complex				✓	✓

Table 8.1: Relations between operations and techniques

The operations need to be related to the techniques that were identified. As already discussed in the techniques chapter (Chapter 6), some of the operations are not wanted by some of the techniques, because they would take too long to performed at runtime. As our focus lies with zero downtime, this will be perceived by the user. Therefore, these operations, which are related to multiple aggregates, are considered operations that you do not want to run in a technique which transforms runtime. As can be seen in Table 8.1, the only operations this will affect are the complex operations on the event store level. These operations are the only operations that are related to multiple aggregates, which can give performance problems.

8.2 Techniques and strategies

The relations between techniques and operations are identified, the techniques and strategies need to be linked to each other. As can be seen in Figure 8.1, for the all the techniques, an application upgrade strategy is needed. The schema evolution techniques also need a data upgrade strategy.

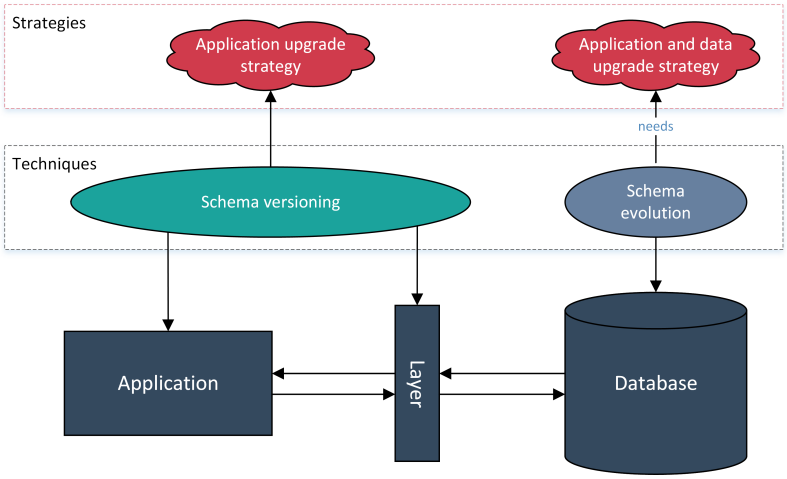

Figure 8.1: Techniques & strategy

This means that, *support multiple versions*, *upcasting* and *lazy transformation* are techniques that only need an application upgrade strategy, to get the technique in place. *In place event store transformation* and *replay event store* are techniques that both need an application and a data upgrade strategy. As a result, many possible combinations exist.

As all the techniques and strategies are described and analyzed, the relationship between them can be described. Appendix A includes two tables that describe them. Table A.1 shows the runtime techniques, combined with the possible application upgrade strategies. Table A.2 shows the non-runtime techniques combined with the application and data upgrade strategies.

In both tables, it is shown whether the combination would result in *zero downtime*, *application upgrade* or *major downtime*. *Application upgrade* is downtime that is caused by the new application upgrade being deployed including the technique. *Major downtime* means that at least the data upgrade or both the application and data upgrade are causing downtime.

8.2.1 Zero downtime combinations

Several combinations score well since they can be realized with zero downtime. For each of the technique these are:

Multiple versions - is possible with zero downtime when applying rolling, big flip or blue-green as the application upgrade strategy.

Upcasting - is possible with zero downtime when applying rolling, big flip or blue-green as the application upgrade strategy.

Lazy transformation - is possible with zero downtime when applying rolling, big flip or blue-green as the application upgrade strategy.

In place event store transformation - is possible with zero downtime when applying rolling, big flip or blue-green as the application upgrade strategy and expand and contract as a data upgrade strategy. Combining in place event store transformation with blue-green would not make any sense.

Replay event store - is possible with zero downtime when applying rolling, big flip or blue-green as the application upgrade strategy and expand and contract or blue-green as a data upgrade strategy.

As the above illustrates, there are multiple different solutions. Worth noticing is that in an application upgrade rolling upgrade, big flip, and blue-green are all strategies result in zero downtime. As a data upgrade strategy, expand contract will always result in a zero downtime. This in contrast to blue green, which not automatically result in a logical combination.

8.3 Final framework

The previous sections combined, result in a final framework, which is visualized in figure 8.2. This framework is the combination of operations, techniques, and strategies and the relationships between them. All the operations, techniques, and strategies are shortly described and explained in tables, which can be found in Appendix B.1, B.2, B.3. In the figure, the turquoise techniques are considered schema versioning, whereas the blue ones are schema evolution.

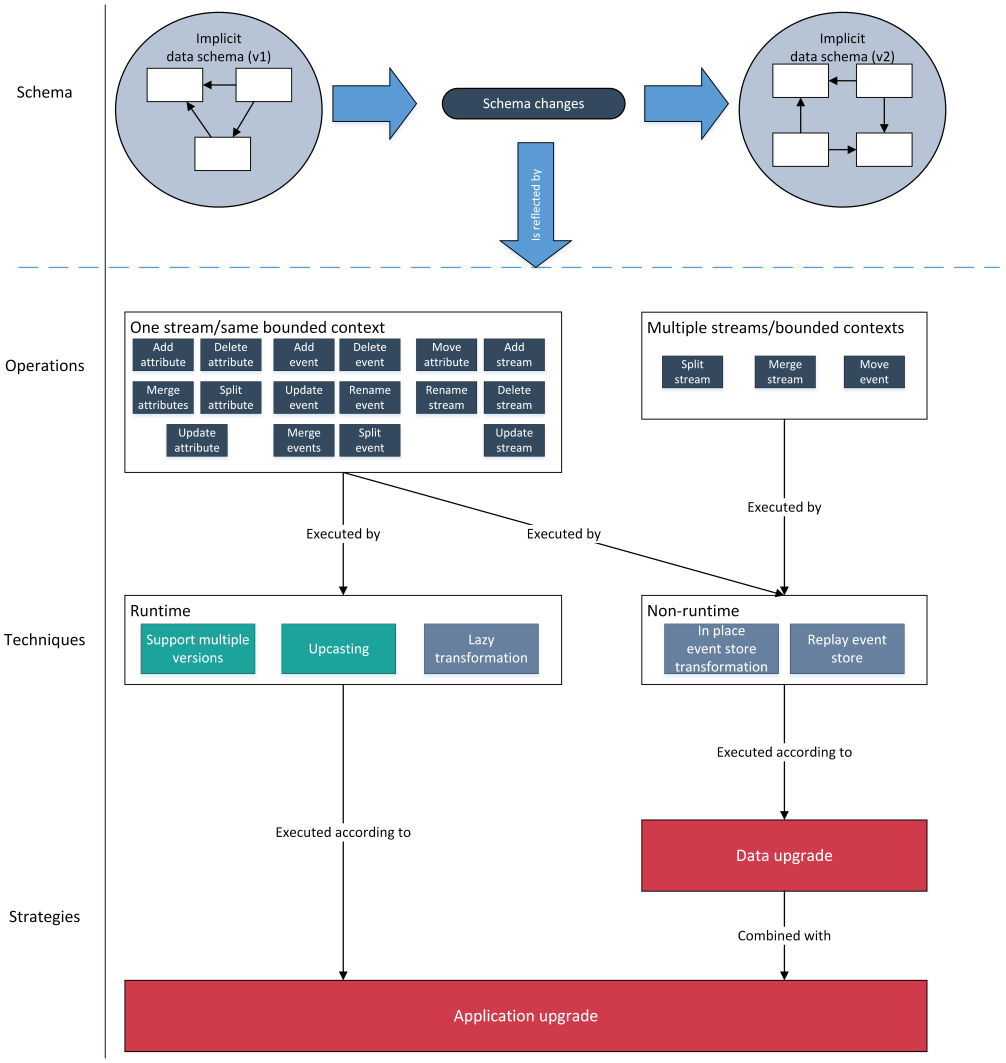

Figure 8.2: Final framework regarding upgrading an event sourced application efficiently

8.3.1 Deciding on technique and strategy

The presented framework shows which technique and strategies you can use. When building your application, several factors influence the scope on which operations to expect, how to handle them and which techniques and strategies you will implement.

First, you have the business need. One of the important discussion points which came up during most expert evaluation interviews and sessions are decisions based on the business aspect. Is it allowed to delete events or even streams from your event store, or does your event store needs to stay immutable? An interesting view on this is to make a difference between *functional immutable* and *technical immutable*. We consider *technical immutable* as being completely immutable, no changes allowed. When your event store is *functional immutable*, you are allowed to make certain changes, which are not directly influencing the event, which is the representation of the saved changing state. You do not change the events themselves, but you can change for example the database platform or the persistency of the event. Another business discussion is the lossless or lossy transformation discussion. Are we allowed to lose information during a data transformation, even when it is (possibly) not relevant anymore? This can be that we do not want them in our active system, but can also be related to have it stored somewhere in the backup of the system.

Secondly, you have environmental factors when deciding on technique and strategy. Maybe some of the data transformation techniques are already implemented in your infrastructure and implementing a new one would be not cost effective. It could also be that your application is not so complex, and only consists of a limited number of streams/bounded contexts. Then you can exclude the multiple stream operations from the framework, which makes the decision process simpler.

In creating this framework, one decision was already made. We wanted to be performance efficient, including zero downtime. Therefore, the relation between multiple streams and runtime techniques was removed in the framework, because we believed this would be too big of an impact on the efficiency of the system.

Figure 8.3 illustrates the several places that can be influenced by the decision-making process. First, you have the translation of your *evolution trigger* to the new implicit schema, which results in the schema changes. In the figure this is marked with the *1*. For example, as a trigger you have the

introduction of the new IBAN bank account number, which needed to be integrated into your application. Several approaches exist to solve this in your schema. This can be done either, by transforming all the current bank account numbers to IBAN in the existing events. Another option is to add a new attribute that consists of the IBAN in all the existing events. A third option is to add a new event including the new IBAN number of the relevant bank account number. This decision needs to be made, based on the existing requirements.

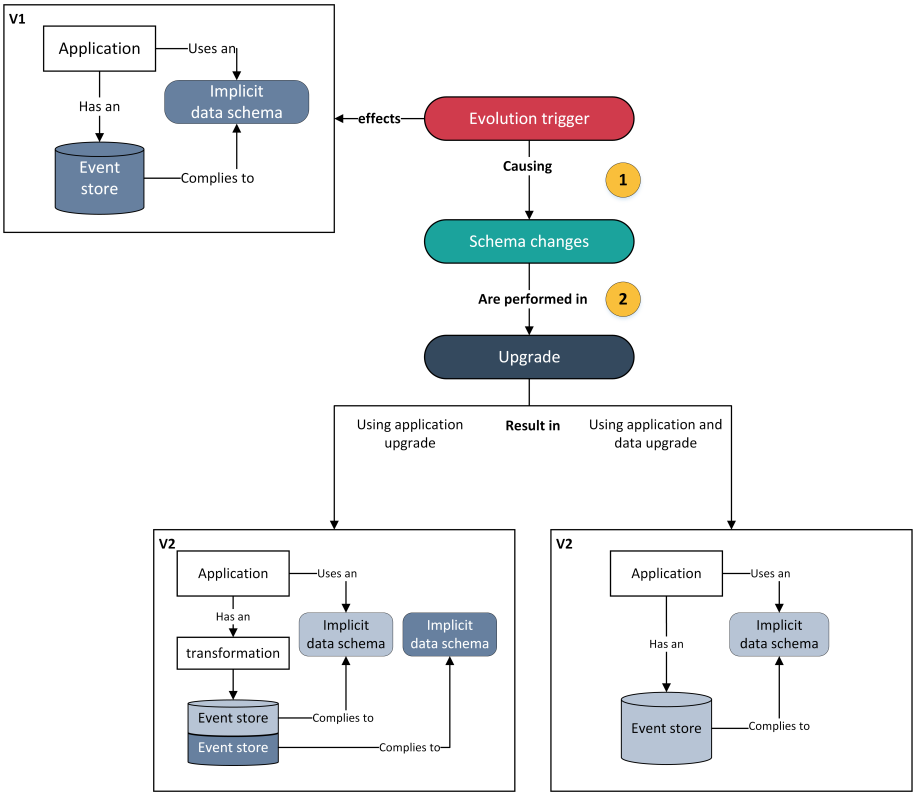

Figure 8.3: Illustrating decision-making based on the framework

After you know how it is affected, it is time to decide on how to upgrade (number 2 in Figure 8.3). As the operations are reflected by the schema change, they will be based on the new schema. However, deciding on which technique and deployment strategy is another important design decision, with different results, as can be seen in Figure 8.3. Certain techniques will result in the left V2 application, having multiple schemas active in your application. The left V2 application is using a *schema versioning* technique, and using only an *application upgrade* strategy. The *schema evolution* technique will result in the right V2 application, using both an *application upgrade and*

data upgrade strategy. One important side note, the lazy transformation is a *schema evolution* technique, which at the start will looking like the left application, but eventually will become the right application.

You can also decide to implement multiple approaches, also referred as a *hybrid approach*. When you have a runtime solution, combined with a non-runtime solution including a data upgrade strategy, you will probably reduce the time when the non-runtime approach is needed. For example, when the first operations are just simple add attribute and merge attribute operations, which are both operations that can be performed by runtime techniques, you do not need the non-runtime technique. However, when the next schema change needs a split stream operation, those operations will probably be included in performing your data upgrade. When the data upgrade is syncing in the background, you reduce the time needed to finalize the data upgrade, as some operations are already (partially) executed.

Chapter 9

Evaluation with experts

Two evaluation phases took place during this research. This chapter describes the evaluations since we thought it would provide valuable insights on what was changed.

9.1 Evaluating the operations overview

The concept overview in Chapter 5 was evaluated in a session with three experts from AFAS Software. The AFAS experts were two lead software architects and the software project manager. These experts have all been working with CQRS and event sourcing for a few years now. During the session two important discussion points came up:

Missing operations - One of the questions that came up during the sessions, was whether certain operations were missing or not. For example, the move attribute/event operation. When creating the overview, the decision was made to leave them out, based on the argument they could be expressed by the other operations.

Lossless vs. lossy - The second important discussion point was, how to handle the fact that when doing some of the operations, you are (possibly) "changing the past". This is especially the case for operations that are not lossless. These questions on how to handle such cases, typically need to be answered by the stakeholders of the applications.

The first discussion point was used when refactoring the operations overview. The second point can be found in Section 8.3.1.

9.1.1 Changes to operation overview the evaluation

After the evaluation, AFAS started to make a framework for the data transformation in their current development of Profit Next. These operations and the cases (or equivalents) were the starting points for their implementation. Based on the new insights, changes were made to the operations overview. The changes were:

The separation between basic and complex operations - One of the discussion points was that not all operations were explicitly present. To improve the overview we decided to make a difference between basic and complex operations (see section 5.2) and to add some operations as described in the next points.

Added move operations - The move operation was explicitly added both to the level of the stream (*move attribute*) and store level (*move event*).

Renaming present on all levels - The rename operation is now present on all the levels. Update (event)type on attribute level was renamed *rename event* and moved to the stream level. *Rename stream* was included at event store level, and rename attribute is represented by the *update attribute* operation on the event level.

Rename create to add - The name of the create operations is changed to add because that is better describing the operations.

9.2 Expert interviews about the framework

To improve end evaluate the final framework, interviews were held with external experts in the field of CQRS & event sourcing. The focus in interviews was to see if this research matches the experience and the knowledge of experts in the field of CQRS & event sourcing. Furthermore, discussing the framework with external experts, involved in one or more other CQRS & event sourcing implementations, would help this research to improve the generalizability.

9.2.1 Interview protocol

For conducting the expert interviews, multiple goals were set:

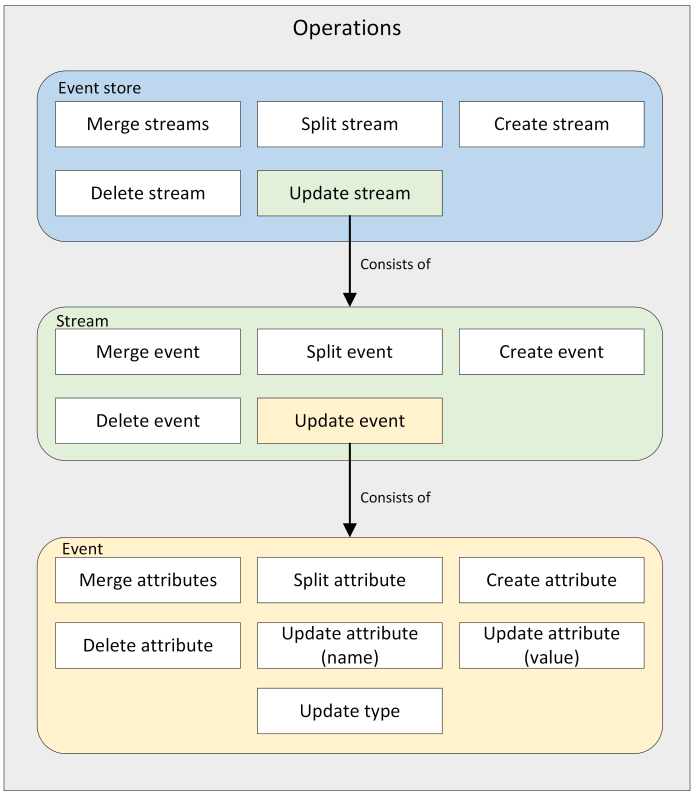

Figure 9.1: Concept operations overview - before the validation session

CQRS & event sourcing upgrade experience - This is the starting point of the interview. This to see what their experiences of upgrading CQRS applications are, and what problems and situations they ran into. This without them already having seen the framework. Furthermore, we want to be able at the end, to place their experiences into the framework. This to see what kind of operations, techniques, and strategies they used when looking at the framework.

Operations overview - We want to know what they did they encounter. Related to the framework, we wanted to know if all the operations were clear, if the operations overview was complete and correct, and if they were missing any operations.

Event store techniques - Do the experts agree on the found existing techniques or are there any techniques missing?

Deployment strategies & final framework - Do they agree on the deployment strategies? Do they see value in the final framework?

An interview protocol was set up to fulfill these goals. This interview protocol can be found in Appendix C.

9.2.2 Execution of the interviews

For conducting the interviews, Dutch experts in the field of CQRS & event sourcing were approached.

Name	Function	Experience
Allard Buijze	CTO, Creator of AXON framework	6,5 years
Dennis Doomen	Consultant in CQRS & event sourcing	6 years
Pieter Joost van der Sande	Engineering Architect	6,5 years

Table 9.1: The interviewed experts

All the interviews were when possible recorded and later summarized where the relevant quotes were picked out. The summaries of the expert interviews can be found in Appendix D.

9.2.2.1 Conclusions based on the interviews

Based on the expert interviews, small changes to the framework were made, and interesting conclusions were drawn.

Naming issues - In all the interviewees, there were small misunderstandings or extra explanation needed because of naming issues. We think this is related to the relatively new concept of event sourcing, where people from different fields (domain-driven design, distributed systems, event-driven architecture) are coming together.

The relation between the level of operation and encounters - All the interviewees said that operations on the level of the store are not common, especially the complex operations.

The specific version of multiple versions is having 'multiple versions" Two interviewees came up with some technique which was related to having multiple versions of an aggregate around, in which *aggregate_v1*, could be input for *aggregate_v2*. This makes it possible to keep some support for the previous versions.

Solving problems in projections - Another thing two interviewees had in common, is that sometimes problems were not solved on the command side of the application, but on the query side. Although not in the scope of the research, it is worth mentioning.

Complete and useful - All the interviewees found the suggested framework useful and complete. All interviewees were interested in the end result. One interviewee mentioned: "we are probably the only one having such a complete picture of this, especially since we also include a lot of scenarios and edge cases".

9.2.2.2 Changes based on the conclusions

Based on the conclusions from the expert interviews, the framework that was presented to the experts was changed. The changes that were made will be explained and motivated.

Lazy upcasting to lazy transformation - One of the interviewees thought *lazy upcasting* was the process of upcasting on a lazy manner, but this was not the correct explanation. That is why one of the experts suggested to rename it to *lazy transformation*, on which we agreed.

Transformation scripts to Transform event store in place - the original name of *transform event store in place* was *transformation script(s) on event store*. The focus in this name should not be on the what, but on the where. As the big difference between this technique and the other techniques is that this is happening on the current event store, the name was changed.

Categorization of deployment strategies was added - As the categorization was not completed before the expert interviews, it was left out. As this proved to be inconvenient during the interviews, it was decided to add it to the final framework. The framework ended up with two interpretations of the strategy (application upgrade or data upgrade), resulting in a lot of extra combinations.

Operations were categorized together - Previously within the framework, four categories for operations were used, to illustrate the specific categories to the experts. As they agreed to the relations between the operations and techniques, the decision was made to put the ones that are linked to the same category techniques, together.

Chapter 10

Discussion

Validity is an important issue regarding doing research. As this research is considered to be empirical research, we need to discuss the several threats to the validity and how we tested or handled them in the best way (Easterbrook et al., 2008).

10.1 Construct validity

Construct validity is about theoretical constructs and their interpretation. For this reason, definitions from literature were used whenever possible. When a definition could not be found, we explicitly stated what our definition or explanation was. Most of the used definitions can be found in the glossary.

10.2 Internal validity

Internal validity is to ensure that results follow from the data. In this research, this is not applicable, as there was no data involved in this research. The only data this research worked with was the operations prototype and the expert interviews. The operations prototype is published on GitHub, so that it can be checked publicly. Summaries of the expert interviews were made, which were checked by the interviewees themselves, to make sure that they agreed with the interpretation.

10.3 External validity

External validity focuses on the justification of the generalization of the results. To do this, expert interviews were conducted the discuss their CQRS & event sourcing experiences and show and discuss the created framework with the experts. This step was explicitly performed to find out if the framework would be as useful and generalizable as we thought.

10.4 Reliability

Reliability has to do with how reliable the results are and their reproducibility. This research was on purpose not set up as a case study but used a broader perspective. Parts of the research can be reproduced, for example the Haskell operations prototype.

Chapter 11

Conclusion

In this chapter, a summary is given of the answers on the research questions. At the end of this chapter, possible future work is presented.

11.1 Main conclusions

Event sourcing and CQRS are relatively new patterns, which cope with challenges from software evolution, as illustrated by Lehman's law.

SRQ 1: *'Which operations are needed to transform events so that they comply to the new event schema?"*

All the operations were described for this subquestion, with the help of brainstorms sessions and a simple Haskell prototype. The overview of operations was later evaluated together with experts, which resulted in a small refactoring. The final overview consisted of three levels (event, stream, store) and two categories, basic or complex operations. These operations are:

Store - basic Add stream, delete stream, rename stream, update stream

Stream - basic Add event, delete event, rename event, update event

Event - basic Add attribute, delete attribute, update attribute

Store - complex Merge streams, split stream, move event

Stream - complex Merge events, split event, move attribute

Event - complex Merge attributes, split attribute

An explanation on what each consists of can be found in Table B.1, in Appendix B.

SRQ 2: *"Which techniques are available to execute data transformation operations?"*

For this subquestion several techniques from non-event store databases (relational, NoSQL) were examined and described, using *schema versioning* and *schema evolution* as categorization. With *schema versioning*, multiple versions of the schema are active in the application. *Schema evolution* means only one schema active in the application and the data is adjusted to this schema. Applying existing techniques to the event store, combining it with the known event store techniques, it resulted in 5 event store data transformation techniques:

Multiple versions - a runtime schema versioning technique. Multiple versions of events are supported everywhere in the application where they are needed.

Upcasting - a runtime schema versioning technique. When retrieving events from the event store, they are upcasted to the latest version of the event.

Lazy transformation - a runtime schema evolution technique. When retrieving events from the event store, they are upcasted to the latest version and these latest versions are send back to the event store, overwriting the old versions.

In place event store transformation a non-runtime schema evolution technique. The existing event store is in place transformed, by for example a transformation script

Replay event store a non-runtime schema evolution technique. The events are replayed to an empty event store, which is set up according to the latest schema. When an event needs transformation, it is done before arriving at the new event store.

These techniques were described including their advantages and disadvantages. Later they were compared and analyzed based on quality attributes. These quality attributes were *functional completeness* (do they support all the operations), *performance efficiency* (is the performance of the technique good), *recoverability* (what happens when the technique fails) and *maintainability* (how is the maintenance of the technique).

SRQ 3: *"What are the existing deployment strategies for upgrading software?"*

For this subquestion an overview was created of the existing deployment strategies. These strategies were visualized and described. It resulted in five deployment strategies:

Deploy on running system - Deploy or run the technique on the running system, which is also known as fast reboot. Will never be a zero downtime solution.

Big flip - Separating your cluster into two groups, which is upgraded one by one. When upgrading, the load balancer redirects everyone to the cluster which is not upgrading. It reduces active servers running your application during deployment.

Rolling upgrade - Similar to big flip, only in this case not two, but multiple groups are used. As a side effect, there are two versions running at the same time.

Expand-contract db deployment - Use ghost tables in the running data store that in the background migrates the current data. Can be referred to as creating a second slot in your data store.

Blue-green deployment - You have two slots, blue and green, which are there for both the application and the database. You deploy the new application or upgrade the data to the not active slot and when finished you make the switch.

All these deployment strategies function as data upgrade strategy. Furthermore, all except expand-contract db deployment, function as an application upgrade strategy.

SRQ 4: *"How can the most appropriate technique be selected and deployed efficiently, given a set data transformation operations?"*

To answer this question, several things needed to be done. The operations (SRQ 1), need to be executed by a technique (SRQ 2), which is then deployed according to an application upgrade and possibly a data upgrade strategy (SRQ 3). The relations between these needed to be identified. First the relation between the operations and technique needed to be identified. The only operations that did not have a good relation with all the techniques, were the complex operations on the level of event store. These did not work well in the runtime techniques (multiple versions, upcasting and lazy transformation).

This was followed by the combinations of techniques and strategy. The runtime techniques only needed an application upgrade strategy and the

other two techniques need both an application and data upgrade strategy. All the combinations were examined, to see what the consequences were and if they would be zero downtime. All these combinations can be found in Appendix A, Table A.2. As an end result for this question, the final framework was created, and several factors influencing the design process were described, to give some guidance in choosing the right approach for your application.

RQ: *"How can an event sourced application efficiently be upgraded in the face of event schema changes?"*

With the help of the framework, you can quickly identify which approaches are there and which would be helpful for your situation. Furthermore, it provides starting points for people that are currently building such an application and that are thinking about a solution for this challenges of the changing event schema.

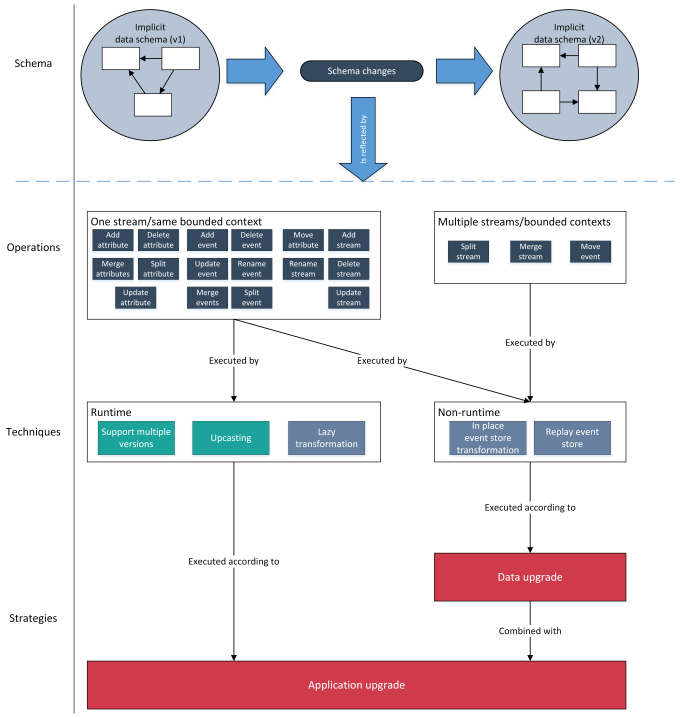

Figure 11.1: Final framework

11.2 Future research

During this research, several possibilities for future work came up, which were out of scope, or would be a follow up for this research. The field of CQRS and event sourcing can use a lot more scientific contributions, to create a scientific base and more interest in these patterns. First, this framework is now evaluated with experts, but there are still options to validate certain parts of this research or create a bigger. For example benchmarking the operations upcasting decision or multiple data transformation stacked onto each other, like upcasting chained over multiple schemas.
Not only benchmarking would be a good addition, but there are also other aspects that might be of interest. For instance, finding out motivations of certain schema changes and ways to prevent them, or let them be simplified.
Another point for future work is that currently only the command side of the CQRS application is discussed for the upgrade. However, updating the event schema can (and often will) also have an effect on the query side of the application, with the projections which use those changed events. This was not included in this research, but it could prove to be an interesting angle.
Another interesting aspect regarding event sourcing, is the option to apply complex event processing on the event store (see related literature, Section 4.3). This many possibilities in the fields of business intelligence and pattern recognition.
Related to the field of model-driven design and code generation, an interesting research topic is to find out whether it is possible to derive and generate the solution for the data transformation, based on the model transformation.

Glossary

Aggregate is combinations of entities, which combined have their own bounded context.

CQRS stands for Command Query Responsibility Segregation, first described by Dahan (2009); Young (2010b). CQRS is an architectural pattern that separates the commands (changing state) from the queries (retrieving data).

Data transformation Update the data from an old (v1) to a new data scheme (v2).

Deployment strategy is an approach to get techniques deployed and/or working.

Domain Driven-Design is often abbreviated to DDD. In DDD you describe your system for both IT and business by mapping business domain concepts into software artifacts.

Event schema defines the implicit data schema of the events in an event store.

Event sourcing is the concept that instead of saving your application state like a normal relational database, saves all the changes to your application state as separate steps, similar to an audit trail (Fowler, 2005).

Event store is a data store for events. An event store consisting of multiple streams. Abbreviated to ES.

Eventually consistent is a weaker form of consistency and means that when no updates are made to the object, the object will eventually have the last updated value (Vogels, 2009).

Implicit schema is the schema that is present in the data in a schemaless data store so the application can interpret the data (Fowler, 2013b).

Operation is a change made to the event schema, resulting in some action.

Projection is the view on state or the state history, created and updated by the projector.

Technique is an approach to execute the operations, with or without touching the data store.

Zero downtime means that no downtime is perceived by the user.

Bibliography

Ameller, D. (2009). *Considering Non-Functional Requirements in Model-Driven Engineering*. Master, Universitat Politècnica de Catalunya.

Axon Framework (2016). Reference Guide Axon Framework reference guide - Event Upcasting. http://www.axonframework.org/docs/2.4/repositories-and-event-stores.html.

Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., and Subramanian, M. (2013). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices.

Brandolini, A. (2013). Introducing Event Storming. http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html.

Brewer, E. (2000). Towards robust distributed systems. In *PODC*.

Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. *Computer*.

Brewer, E. A. (2001). Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55.

Callaghan, M. (2010). Facebook - Online Schema Change for MySQL. https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/.

Cattell, R. (2011). Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27.

Chandy, K. M. (2009). Event Driven Architecture. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1040–1044. Springer US.

Clifford, J. (1982). A model for historical databases. *Information Systems Working Papers Series, Vol.*

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Curino, C., Moon, H. J., Deutsch, A., and Zaniolo, C. (2013). Automating the database schema evolution process. *VLDB J.*, 22(1):73–98.

Curino, C., Moon, H. J., and Zaniolo, C. (2008). Graceful database schema evolution: the PRISM workbench. *PVLDB*, 1(1):761–772.

Dahan, U. (2009). Clarified CQRS. Dec. 2009. http://www.udidahan.com/2009/12/0.

de Jong, M. (2015). *Zero-Downtime SQL Database Schema Evolution for Continuous Deployment*. Master, TU Delft.

de Jong, M. and van Deursen, A. (2015). Continuous deployment and schema evolution in SQL databases. In *Proceedings of the Third International Workshop on Release Engineering*, Firenze.

Domínguez, E., Lloret, J., Rubio, A. L., and Zapata, M. A. (2008). MeDEA: A database evolution architecture with traceability. *Data Knowl. Eng.*, 65(3):419–441.

Dumitras, T. and Narasimhan, P. (2009a). No downtime for data conversions: Rethinking hot upgrades. Technical report, Carnegie Mellon University, Pittsburgh.

Dumitras, T. and Narasimhan, P. (2009b). Why Do Upgrades Fail and What Can We Do about It? In Bacon, J. and Cooper, B. F., editors, *Middleware 2009, ACM/IFIP/USENIX, 10th International Middleware Conference, Urbana, IL, USA, November 30 - December 4, 2009.*, volume 5896 of *Lecture Notes in Computer Science*, pages 349–372, Urbana, IL, USA. Springer.

Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2008). Guide to Advanced Empirical Software Engineering. *Guide to Advanced Empirical Software Engineering*, pages 285–311.

Elbushra, M. and Lindström, J. (2014). Eventual consistent databases: State of the art. *Open Journal of Databases (OJDB).*

Erb, B. and Kargl, F. (2015). A conceptual model for event-sourced graph computing. In Eliassen, F. and Vitenberg, R., editors, *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 352–355. ACM.

Etzion, O. (2009). Complex Event & Complex Event Processing (CEP). In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 411–413. Springer US, Boston, MA.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2005). Event sourcing. http://martinfowler.com/eaaDev/EventSourcing.html.

Fowler, M. (2010). BlueGreenDeployment. http://martinfowler.com/bliki/BlueGreenDeployment.html.

Fowler, M. (2013a). Introduction to NoSQL.

Fowler, M. (2013b). Schemaless Data Structures. http://martinfowler.com/articles/schemaless/.

Fowler, M. (2016). YOW! Nights - Event Sourcing. https://www.youtube.com/watch?v=aweV9FLTZkU.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59.

Guelen, J. (2015). *Informed CQRS design with continuous performance testing*. Master thesis, Utrecht University.

Hartung, M., Terwilliger, J. F., and Rahm, E. (2011). Recent Advances in Schema and Ontology Evolution. In Bellahsene, Z., Bonifati, A., and Rahm, E., editors, *Schema Matching and Mapping*, Data-Centric Systems and Applications, pages 149–190. Springer.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105.

Hick, J.-M. and Hainaut, J.-L. (2006). Database application evolution: A transformational approach. *Data Knowl. Eng.*, 59(3):534–558.

Hohpe, G. (2006). Programming without a call stack-event-driven architectures. *Objekt Spektrum*.

Humble, J. and Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Professional.

ISO/IEC (2011). ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

Jacob, S. A. and Furgerson, S. P. (2012). Writing interview protocols and conducting interviews: Tips for students new to the field of qualitative research. *The Qualitative Report*, 17(42):1–10.

Jensen, C. S. and Snodgrass, R. T. (2009). Temporal Database. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 2957–2960. Springer US.

Kabbedijk, J., Jansen, S., and Brinkkemper, S. (2012). A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software. In *Proceedings of the 17th European Conference on Pattern Languages of Programs*, Irsee.

Korkmaz, N. (2014). *Practitioners' view on command query responsibility segregation*. Master thesis, Lund University.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Mellor, S. J., Clark, A. N., and Futagami, T. (2003). Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall.

Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, 2.

Niblett, P. (2009). Event Transformation. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1064–1068. Springer US.

Nichter, D. and Baron, S. (2016). Percona Toolkit documentation of pt-online-schema-change. https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html.

Noach, S. (2014). Openark Kit. https://code.google.com/archive/p/openarkkit/.

Penney, D. J. and Stein, J. (1987). Class modification in the GemStone object-oriented DBMS. In Meyrowitz, N. K., editor, *ACM SIGPLAN Notices*, volume 22, pages 111–117, Orlando, Florida, USA. ACM.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393.

Roddick, J. F. (2009). Schema Evolution & Schema Versioning. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 2479–2481, 2499–2502. Springer US, Boston, MA.

Sadalage, P. and Fowler, M. (2012). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley.

Sato, D. (2014). ParallelChange. http://martinfowler.com/bliki/ParallelChange.html.

Scherzinger, S., Klettke, M., and Störl, U. (2013). Managing Schema Evolution in NoSQL Data Stores. In Green, T. J. and Schmitt, A., editors, *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy*.

Tan, L. and Katayama, T. (1989). Meta Operations for Type Management in Object-Oriented Databases. In *DOOD*, pages 241–258.

Traub, D. and Simmons, C. (2011). Using an RDBMS as event sourcing storage. http://stackoverflow.com/a/7065225.

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40–44.

Young, G. (2010a). CQRS and CAP Theorem. http://codebetter.com/gregyoung/2010/02/20/cqrs-and-cap-theorem/.

Young, G. (2010b). CQRS and Event Sourcing. Feb. 2010. http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing.

Young, G. (2013). Querying Events at Code Mesh London 2013. https://www.youtube.com/watch?v=BWOaUT9T-jA.

Young, G. (2016). A Decade of DDD, CQRS, Event Sourcing - Domain-Driven Design Europe 2016. https://www.youtube.com/watch?v=LDW0QWie21s.

Zdonik, S. B. (1986). Version Management in an Object-Oriented Database. In Conradi, R., Didriksen, T., and Wanvik, D. H., editors, *Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway, June 16-18, 1986*, volume 244 of *Lecture Notes in Computer Science*, pages 405–422, Trondheim, Norway. Springer.

Appendix A

Tables techniques & deployment strategies combined

Techniques	Application	Zero downtime	Description
Multiple versions	Running system	Application upgrade	Short downtime during installation
	Big flip	Zero downtime	reduced hardware pool, possibly catch up phase between data stores when done.
	Rolling	Zero downtime	Users can not turn back to old version, possibly catch up phase when new server(s) are added to finished pool.
	Blue-green	Zero downtime	Kind of big flip, with rollback and no experienced reduced hardware pool.
Upcasting	Running system	Application upgrade	Short downtime during installation
	Big flip	Zero downtime	reduced hardware pool, possibly catch up phase between data stores when done.
	Rolling	Zero downtime	Users can not turn back to old version, possibly catch up phase when new server(s) are added to finished pool.
	Blue-green	Zero downtime	Kind of big flip, with rollback and no experienced reduced hardware pool.
Lazy transformation	Running system	Application upgrade	Short downtime during installation
	Big flip	Zero downtime	reduced hardware pool, possibly catch up phase between data stores when done.
	Rolling	Zero downtime	Users can not turn back to old version, possibly catch up phase when new server(s) are added to finished pool.
	Blue-green	Zero downtime	Kind of big flip, with rollback and no experienced reduced hardware pool.

Table A.1: Runtime techniques combined with application deployment strategies

Table A.2: In place event store transformation and replay event store combined with application deployment strategy and data upgrade strategy.

Techniques	**Application upgrade**	**Data upgrade**	**Zero downtime**	**Description**
In place ES	Running system	Running system	Major downtime	Major downtime, because both application and data(schema) update is done on running system
In place ES	Running system	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Running system	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Running system	Expand contract	Application upgrade	If application deployed in same time as contract phase, only downtime during the initiation of the expand phase and the catch up phase during the contract phase
In place ES	Running system	Blue-green	Application upgrade	When application is deployed after blue-green the data upgraded finished, pure downtime for deploying application and possibly small catch-up phase
In place ES	Big flip	Running system	Major downtime	Major downtime, because both application and data(schema) update is done on running system

In place ES	Big flip	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Big flip	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Big flip	Expand contract	Zero downtime	
In place ES	Big flip	Blue-green	Not applicable	Not applicable, data upgrade to parallel version (b/g), and then do the in place upgrade
In place ES	Rolling	Running system	Major downtime	Major downtime, because both application and data(schema) update is done on running system
In place ES	Rolling	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Rolling	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Rolling	Expand contract	Zero downtime	
In place ES	Rolling	Blue-green	Not applicable	Not applicable, data upgrade to parallel version, and then do the in place upgrade

In place ES	Blue-green	Running system	Major downtime	Major downtime, because both application and data(schema) update is done on running system
In place ES	Blue-green	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Blue-green	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
In place ES	Blue-green	Expand contract	Zero downtime	
In place ES	Blue-green	Blue-green	Not applicable	Not applicable, data upgrade to parallel version, and then do the in place upgrade
Replay ES	Running system	Running system	Major downtime	Major downtime, because both application and data(schema) update is done on running system
Replay ES	Running system	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Running system	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Running system	Expand contract	Application upgrade	Instead of only building ghost tables for changed stuff, do it for the complete event store and replay

Replay ES	Running system	Blue-green	Application upgrade	
Replay ES	Big flip	Running system	Major downtime	Because data upgrade is done fast, major downtime
Replay ES	Big flip	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Big flip	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Big flip	Expand contract	Zero downtime	Reduced hardware, replay to the same DB
Replay ES	Big flip	Blue-green	Zero downtime	Reduced hardware, replay to parallel blue/green version
Replay ES	Rolling	Running system	Major downtime	Because data upgrade is done fast, major downtime
Replay ES	Rolling	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Rolling	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application

Replay ES	Rolling	Expand contract	Zero downtime	Reduced hardware
Replay ES	Rolling	Blue-green	Zero downtime	Reduced hardware, replay to parallel blue/green version
Replay ES	Blue-green	Running system	Major downtime	Because data upgrade is done fast, major downtime
Replay ES	Blue-green	Big flip	Major downtime	Data upgrade is done big flip, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Blue-green	Rolling	Major downtime	Data upgrade is done rolling, which means the users which data is on the servers that are upgraded, can't use the application
Replay ES	Blue-green	Expand contract	Zero downtime	Replay to same event store
Replay ES	Blue-green	Blue-green	Zero downtime	Replay to parallel blue/green version

Appendix B

Explanation tables operations, techniques and strategies

Level	Complexity	Operation	Description
Event	Basic	Add attribute	Attribute is added to an event
		Delete attribute	Attribute is deleted from an event
		Update attribute	Attribute is updated, can be both the name or value(type)
	Complex	Merge attributes	Two attributes are combined to one attribute
		Split attribute	One attribute is split into two attributes
Stream	Basic	Add event	A new event type is added to the stream
		Delete event	An event(type) is deleted from the stream
		Rename event	An event(type) is renamed
		Update event	An event is updated by one or multiple event operation(s)
	Complex	Merge events	Multiple events are combined to one
		Split event	One event is split into multiple
		Move attribute	One attribute is moved from one event type to another
Store	Basic	Add stream	A new stream is added to the store
		Delete stream	A stream is deleted from the stream
		Rename stream	A stream is renamed
	Complex	Merge streams	Multiple streams are combined to one stream
		Split stream	One stream is split into multiple streams
		Move event	An event is moved from one stream to another stream

Table B.1: All the operations

Schema versioning/evolution	**Technique**	**Description**
Schema versioning	Multiple versions	Multiple versions of events are supported everywhere they are needed
	Upcasting	When retrieving events from the event store, they are "upcasted" to the latest version
Schema evolution	Lazy transformation	When retrieving events from the event store, they are "upcasted" to the latest version and these latest versions are send back to the event store, overwriting the old versions
	Inplace ES transformation	The current event store is in place transformed, by for example a transformation script
	Replaying ES	The events are replayed to an empty event store, which is set up according to the latest schema. When an event needs transformation it is done before arriving at the new event store

Table B.2: All the event store techniques

Table B.3: All the deployment strategies

Strategy	**Fit**	**Description**
Deploying on running system	Application & Data upgrade	Just deploy/run the technique on the running system, also known as fast reboot. Will never be a zero downtime solution.
Big flip	Application & Data upgrade	Separating your cluster iton two groups, which is upgraded one by one. When upgrading, the load balancer redirects everyone to the cluster which is not upgrading. It reduces active servers running your application during deployment.
Rolling upgrade	Application & Data upgrade	Similar to big flip, only in this case, not two but multiple groups are used. As a side effect, there are two versions running at the same time.
Expand-contract	Data upgrade	Use ghost tables in the running data store that in the background migrates the current data. Can be referred to as creating a second slot in your data store.
Blue-green	Application & Data upgrade	You have two slots, blue and green, which are there for both the application and the database. You deploy the new application or upgrade the data to the not active slot and when finished you make the switch.

Appendix C

Expert interview: interview protocol

This is the interview protocol, for the interviews with experts in the field of CQRS & event sourcing. When possible we will also record the interview.

C.1 Protocol

Introduction

- Introduction to AMUSE & this research project
- Explain the reason for the interview
- Ask to let him introduce itself a bit more (background, current role)

Experience

- Years of experience with CQRS/event sourcing
- Ask to number of projects were you experienced upgrading
- What were the solutions (techniques, strategy) you used when upgrading the event store

Operations

- Show the expert the operations overview
- Are the operations clear to you?
- Do you agree all the suggested operations exists?
- Do you miss an operation or do you see any improvements?

Event store techniques

- Show the expert the techniques images
- Explain schema versioning/evolution
- Are all the event store techniques clear/correct?
- Do you miss a technique?
- Do you see any improvements?

Deployment strategies

- Show the expert the framework image, explain the deployment strategies
- Are all the deployment strategies clear/correct?
- Do you miss a strategy?

Framework

- Is the mapping between operations and techniques correct? Did you experienced the same?
- Is the mapping between techniques and deployment strategies correct?
- What do you think of such an image?
- What could be improved?

End discussion

- Do you have any suggestions/open remarks?

C.2 Preliminary figures used during interviews

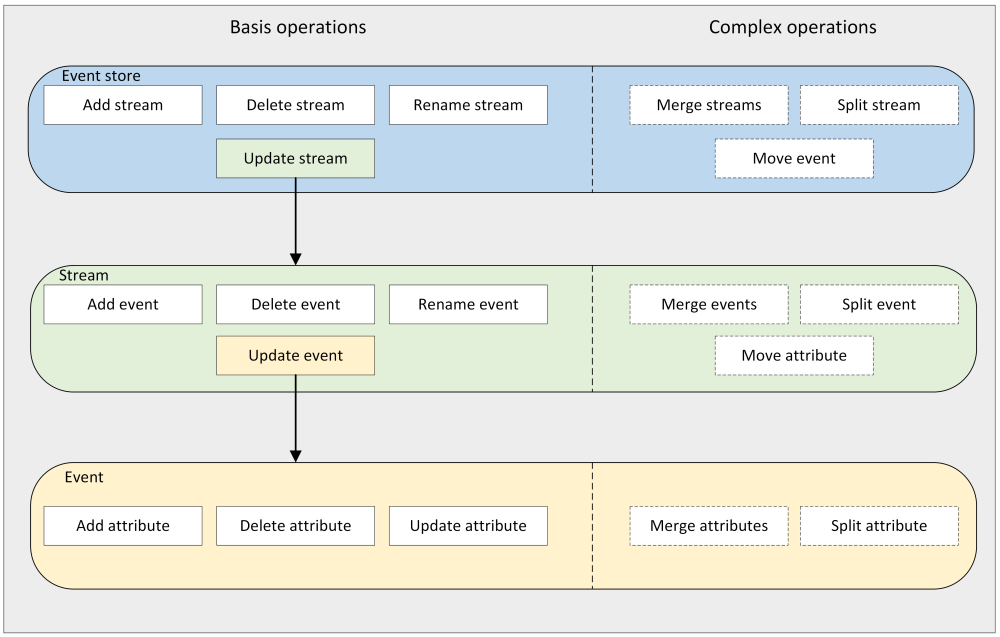

Figure C.1: Expert interviews: discussed operations overview

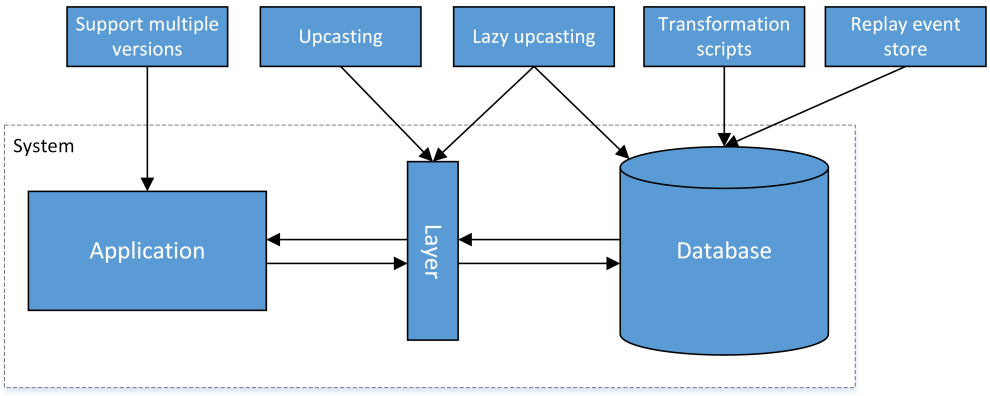

Figure C.2: Expert interviews: discussed techniques overview

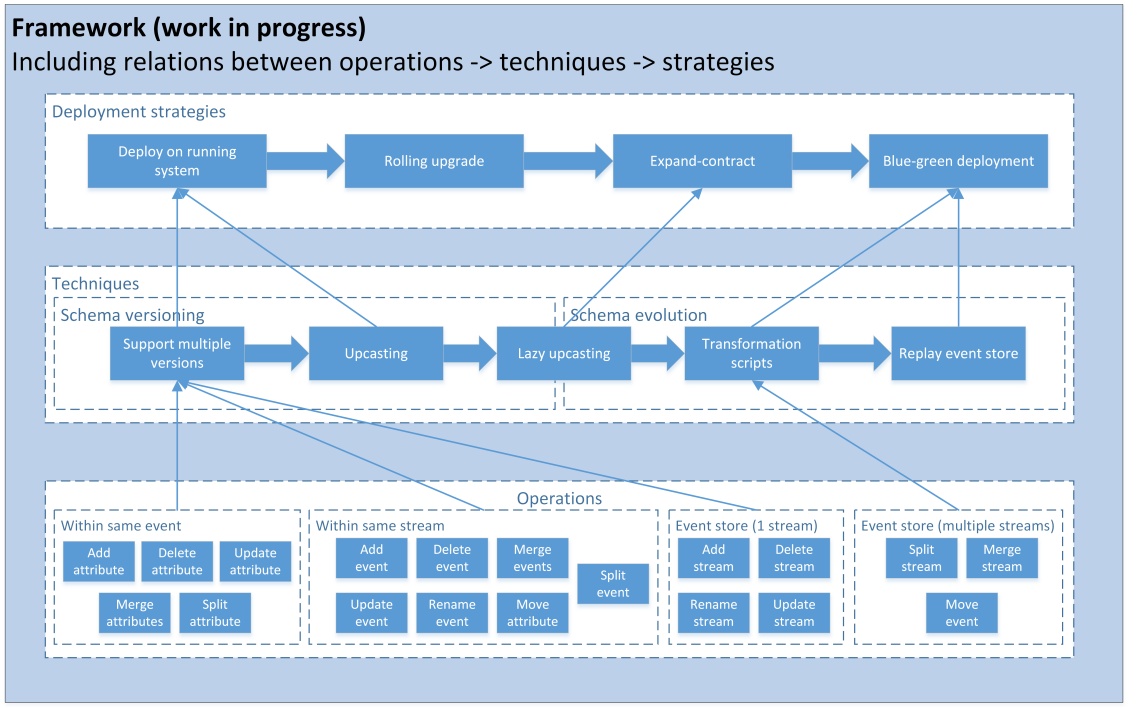

Figure C.3: Expert interviews: discussed solution

Appendix D

Expert interview: summaries

D.1 Allard Buijze

Allard Buijze has 6,5 years experience with CQRS and event sourcing. He started a small project for himself that grew into the Axon framework, which is an open source framework. The company he currently works, Trifork, is helping customers with solutions, like Axon. Through Trifork he is/was involved with several implementations regarding CQRS and event sourcing.

When introducing my topic, Allard states that an event sourced application is something that needs upfront thought, with the help of approaches like event storming. This does not necessary makes agile development and event sourcing not a good match, however event sourcing does require more upfront thinking then many are used to when doing an agile project.

D.1.1 Operations

Regarding the completeness of the overview, the only thing he saw as missing is duplicate event, being the operation that will probably be used by some other complex operation. He does say that some operations are highly unlikely. The complex operations on the level of event store he just experienced only once. That situation was related to wrong bounded context design, because of which change was needed. He was shortly consulted by this client and advised to do something about it. They did nothing to the schema and later the project was killed. One of his other remarks regarding operations

is that delete stream is something that is not a regular operation. A stream will be practically always turned off, archived or deprecated, but never really thrown away. When doing this, special caution is needed, as the removed data may remove context from the other streams.

Furthermore, he says there is a relation between the level of operations and numbers of times he ran into them. He called it exponential, with level of attributes being the most and streams the least number of times.

D.1.2 Techniques

Regarding the techniques, upcasting is the technique which is supported in Axon. Sometimes developers prefer to solve it in the model and do not want to use upcasting. Allard also has experience with transformation scripts. They do not use it for changing events themselves, but for things like the changing the AggregateId and other aspects not directly related to the event content.

Talking about the techniques, he thinks the name of lazy upcasting is not correct. He says this refers to upcasting being done lazy, not doing the transformation lazy, with combining upcasting with writing it to the event store afterwards. He wishes nobody uses or is planning to use the "support multiple versions" technique, as it becomes unneeded complex.

One of the techniques he recently came across was to put the old classes, like old versions of classes in packages. The new classes have a reference to this old class. Kind of transformation from old to the new class. Which such an approach you can also keep (some) support for the old version of the application.

D.1.3 Framework

When showing the complete framework including deployment strategies, Allard sees a big challenge in the rolling update, having two event types up and running at the same time. He sees the need of short to zero downtime, as in his experience, when doing a new release/upgrade the time window they get is becoming smaller and smaller. Though often companies can do a little time without them outside the office hours.

As a conclusion he thinks the overall framework is quite complete. He got some new ideas because of the discussion we had regarding several points,

so it ended up being useful for the both of us.

D.2 Dennis Doomen

Dennis has more than six years experience with CQRS and four years with event sourcing. He worked for the same customer since 2010, but is an active speaker, and within his company he also hears stuff about other implementations. Within the Netherlands Dennis only encountered a handful of CQRS & event sourcing implementations.

D.2.1 Operations

Dennis says technical you should never delete something. When you are doing something like that, it is always just deprecating the stream or archiving/backup the stream/store. This is because when deleting it, it can have legal implication. Update attribute for Dennis would almost always mean adding this updated attribute as a new attribute to the event.

For his current project, they only have experienced all operations on event level and the basic operations on stream level. They can not do complex operations on the stream level as they do not (always) have the entire stream because of the distributed character of the system.

D.2.2 Techniques

When explaining the techniques and the original names, they were not very clear to Dennis. This because some of the names have a different meaning for him or he uses a different name for the same technique. He states that what in the presented research is called upcasting, is called conversion by him and most of the DDD world. Furthermore, replay event store is for him more related to building up the aggregates or projections, not to the explained technique. He finds lazy upcasting as a theoretical technique, which he hopes nobody is going to use, because of the high complexity regarding transactions/rewriting on the event store. Something else he mentioned is that sometimes they solve operations not by applying a technique, but solve it in the projectors.

Dennis prefers to keep the event store immutable with all the benefits that it gives you. All though he never encountered it, Dennis can imagine that after a few years, when you have a few upcasting steps into your application, you will do a one-time transition with a schema evolution technique like transformation scripts or replay event store, for performance reasons.

One interesting technique Dennis heard of, is a sort of upcasting between aggregates. Aggregate v1 reads the v1 events and is used as input for aggregate v2. With this approach, you can keep multiple domain versions alive.

D.2.3 Framework

For his current project, they are doing deployment on running system and blue-green deployment. They use blue-green to have two applications next to each other, not to run any data transformation technique. For him this seemed like a validation of the fact that you can do the operations with runtime solutions and not using the techniques related to schema evolution.

He literally said that we are probably the only one having such a complete picture of this, with also including a lot of scenarios/edge cases he never experienced and not hoped to see. The biggest thing he though missing was the query side/projections related to upgrading such a projection, but this was left out of the scope of this research.

D.3 Pieter Joost van de Sande

Pieter Joost started with CQRS around 6,5 years ago. For him, it started with building a framework, called NCQRS[1]. Since then he worked on several projects building CQRS and event sourcing applications. In his job, working at Double Dutch, they are stepping away from event sourcing, going to an event-driven architecture. In this current project, they are using Apache Samza[2], which was originally developed at LinkedIn.

He thinks event sourcing is not suitable for everybody. You first need to be able to change your mindset to event sourcing. Furthermore, only a few

[1]NCQRS can be found on https://github.com/pjvds/ncqrs
[2]Apache Samza can be found at https://samza.apache.org/

application have the need for scalability or all the events from analyzing purposes. As when you start using it, you get extra problems with upgrading. As event sourcing needs event migration, opposed to the state migration in normal systems. Furthermore, as an audit log, he experienced many companies do not need event sourcing for auditing purposes. A simple log that is created somewhere else, is often also sufficient.

D.3.1 Operations

He agrees with all the operations, which are comparable with the operations which exist in Kafka, the streaming layer of Samza. Pieter Joost states that the number of encounters with operations is also dependent on the granularity of events and streams used in your application.

One of the cases he encountered, was in a hospital. In the hospital, they needed to anonymize and threw away privacy records after a few months, because of regulations. This way it could still be used for research, but not being linked to a person anymore.

D.3.2 Techniques

When discussing the techniques, he prefers techniques which also update the event store, so you do not have to do it each time. Regarding upcasting, he said "why shouldn't you write it back to the event store, so you do not need to do it the next time", which we named lazy transformation. A downside to lazy transformation he mentioned, is that you do not know when lazy transformation transformed the complete event store, so that you can clean up the transformation code.

Regarding upcasting, he also named some other downsides. When upcasters have external dependencies, you need to guarantee that external service during the complete duration of the system. Furthermore, upcasting can be different across multi technologies, when multiple services running on different technologies need to read the event store. He prefers doing in place event store transformation, as that is the shortest solution to an upgraded event store.

Another thing which came up in discussion that there is a difference in perspective of being immutable, between the different experts. Pieter Joost suggested making a difference between being technical immutable and

functional immutable. Technical immutable means staying completely immutable. Functional immutable is a form in which the events themselves, which represented what happened, stay immutable. But for example the store is moved to another database, or the events represent the same thing, only in a different way. Then you do not (have to) change what happened, but though an operation is performed. Another reason for this is the agile way of work. When developing, a consequence is that your schema will change, which are reflected by the operations. When allowing being functional immutable, you do not have to perform any dirty hacks to your current events.

D.3.3 Framework

The discussion ended with a short explanation about the complete framework. He found the overall framework and different aspects very interesting, including the interesting discussions. He said we should look into the possibility of presenting some of this work on an upcoming developers conference.