Evaluating fine-grained events for an Event Sourcing proof-of-concept

HENRIK NGUYEN
Evaluating fine-grained events for an Event Sourcing proof-of-concept

Henrik Nguyen

Thesis submitted to
KTH Royal institute of technology

Masters in Software Engineering of Distributed Systems
School of Electrical Engineering and Computer Science

Supervisors: Anne Håkansson, Alex Gorunescu
Examiner: Mihhail Matskin
Date: 10 June, 2019
Stockholm, Sweden
Abstract

Data conversion for evolving events in an Event Sourcing System is a complex issue and needs to be maintainable. There are suggested ways handling data conversion today which combine different methods into a framework. However, there is a lack of exploration of different and alternative methods to handle the complicated matter.

This thesis explores data conversion with fine-grained events. The purpose is to explore methods and broaden knowledge for handling data conversion while using attribute driven events called fine-grained events. The goal was to build a proof-of-concept that preserves the attributes reliability and availability and can handle data conversion of these specific events.

The results found by using fine-grained events are a decrease in terms of system complexity and a proof-of-concept that maintains the desired attributes.

keywords: Event sourcing, data conversion, proof-of-concept, fine-grained events
Sammanfattning


Resultaten som hittas genom att använda finkorniga händelser är en minskning av systemkomplexiteten och ett bevis på koncept som upprätthåller de önskade egenskaperna.

Nyckelord: Event sourcing, data conversion, proof-of-concept, fine-grained events
Acknowledgements

I would like to thank my supervisors Anne Håkansson and Alex Gorunescu for their support and involvement in this thesis. I would like to especially thank Alex Gorunescu for the guidance and insight he has provided throughout the project. A special thank you to Vizrt and Robert Olsson for their support.
Contents

1 Introduction ................................................................................................................................................................. 1
  1.1 Background .................................................................................................................................................. 1
  1.2 Problem statement ............................................................................................................................................. 2
  1.3 Purpose ............................................................................................................................................................ 2
  1.4 Goal ................................................................................................................................................................. 3
  1.5 Societal benefits and sustainability .............................................................................................................. 3
  1.6 Method ............................................................................................................................................................ 4
  1.7 Delimitation .................................................................................................................................................... 4
  1.8 Contribution ................................................................................................................................................... 5
  1.9 Disposition .................................................................................................................................................... 5

2 Concepts and system design ......................................................................................................................................... 7
  2.1 Domain Driven Design ...................................................................................................................................... 7
  2.2 Command Query Responsibility Segregation .............................................................................................. 9
  2.2.1 CQRS and eventual consistency ........................................................................................................... 12

3 Event Sourcing system ................................................................................................................................................. 13
  3.1 Event Sourcing ................................................................................................................................................. 13
  3.1.1 Event store ............................................................................................................................................... 15
  3.2 Effects of varying granularity of events ......................................................................................................... 17
  3.3 Framework for handling data conversion ....................................................................................................... 17
  3.3.1 Evolving events techniques .................................................................................................................... 18
  3.3.2 Techniques for handling complex operations ......................................................................................... 19
  3.3.3 Deployment strategies .......................................................................................................................... 20

4 Research method ........................................................................................................................................................... 23
  4.1 Research steps .................................................................................................................................................. 23
  4.2 Qualitative research .................................................................................................................................... 23
  4.2.1 Literature study .......................................................................................................................................... 23
  4.2.2 Practical study .......................................................................................................................................... 24
  4.2.3 Validity ...................................................................................................................................................... 25
Chapter 1

Introduction

Since the growth of the internet, the amount of data and information has also increased. One reaction from this advancement is data mining, the practice of analyzing preliminary data as a technique to discover patterns, thus establishing relationships to solve problems [1]. In software development, relational databases [2] are often used as a method to store data. Relational databases follow the CRUD (Create, Read, Update, Delete) model [3], which means that the database is modified using typical CRUD operations and commands, actions that do not consider the use of old data.

1.1 Background

Martin Fowler first introduced the concept of Event Sourcing in 2005 [4]. However, we can trace the idea back to the community of Domain-driven Design [5]. The concept of Event Sourcing is to have each event that resulted from a change to an entity stored in a database. The events are replayed as a chain of events that results into the current state of the entity, much like how every transaction in a bank account are shown to explain and justify the balance of the account.
1.2 Problem statement

A possible disadvantage in persisting current state of data with relational databases is not knowing what the previous state of an entity was, that is to say, how the data reached its current state and why, a problem for which Event Sourcing presents a solution to.

Event Sourcing introduces complex problems such as Data conversion, eventual consistency and projection migration challenges. Consequently, not many studies on Event Sourcing tackle those issues, which has discouraged many developers and researchers from implementing and studying such a system. There is some proposed solution that solves Data conversion provided by Overeem et al. and Greg Young.

Specific requirements define what an event is such as its intent and immutability. However, there is no clear definition of how complex an event should be and how to convert such information. The main question the thesis seeks to answer is: How fine-grained should events be in an Event Sourcing system?

To answer the question, we need to know whether the choice of event granularity can affect the complexity of the system. To further explore the question, a proof-of-concept is developed. A derivative of the previous question is whether or not alternative solutions can be found to handle the same problems.

1.3 Purpose

The purpose of this thesis is to broaden the knowledge on an event-driven storage concept, the concept known as Event Sourcing and assist in the pursuit of creating more sustainable data management.

Current state data storing lacks the element of proof when it comes to how the current state was reached. Despite an increase of popularity of Event Sourcing, we can still see that CRUD is still the model of choice in most implementations. There is also a lack of discussion and evaluations of the concerns surrounding Event Sourcing, one of the topics being data conversion.

The intended audience of this paper are people who would want to spend time on developing a system that uses Event Sourcing with fine-grained events. If one has the incentive to learn more about Event Sourcing and some of the complications of Event Sourcing, they will benefit from this thesis. Exploration of data conversion methods and event-granularity would
1.4 GOAL

thus be beneficial and also aligns with the purpose of this thesis.

1.4 Goal

The main goal of this thesis is to develop an Event Sourcing proof-of-concept. To help reach the main goal, a set of sub-goals were defined and presented below:

• Explore data conversion methods provided in papers written by Young, Overeem et al. [6, 8] and other leading experts on the topic of Event Sourcing to further the understanding of Event Sourcing.

• Present and evaluate the resulting architecture and event granularity based on qualitative research.

1.5 Societal benefits and sustainability

As the amount of generated and stored data increases risks, ethics and sustainability have to be considered in terms of how data is processed and the use cases of the collected information.

For the ethical aspect that requires consideration is the privacy of information and especially the study of behaviour based on the information we gain from tracking and analyzing data patterns, such information should not be maliciously exploited. There are even rules and regulations set by the European Commission around such activities, namely the General Data Protection Regulation also known as GDPR [9].

This thesis does not focus on how the data used with malicious intent can be avoided. Instead, the focus lies in how we can store and represent all of the events to the current state with a proper solution for data conversion that is scalable.

In regards to sustainability, large amounts of data are being lost every day and deemed not necessary when deleted and updated. By eliminating the possibility of removing and overwriting data, we can provide a complete audit log of every state the data has ever gone through since its primary storage and the log can also prove as a beneficial tool for debugging. A drawback of building a complex system such as an Event Sourcing system is the maintenance of the system. A solution of such drawback is well-
structured system design, well-written documentation and version control of the implementation.

1.6 Method

The most common methods in terms of scientific research are the qualitative and quantitative method and the application of each particular method being non-numerical and numerical [10, p.3].

Qualitative methodology targets the meaning, opinions and behaviour to reach a hypothesis, while the purpose of the Quantitative research method is to test the theories and hypothesis by measuring variables to verify or falsify them. Both methods can be used in research, as they reinforce one another and improves the quality of the research by proving correctness. This method of researching is called Triangulation.

The most suited method for this thesis is Triangulation. The thesis demands theoretical background research to build the system and experiments have to be carried out as a method to evaluate the system.

There are several angles to be considered when it comes to the premise of a study. Given the nature of the thesis, the philosophical presumption is a combination of positivism and Interpretivism.

Interpretivism seeks to understand phenomena, by exploring higher dimensions of richness, complexity and depth inductively.

Positivism assumes the self-reliance between the observer and the system being observed. The paradigm concludes what is observed to either approve or dismiss the hypothesis [10].

1.7 Delimitation

The thesis does not seek solutions for the issue of eventual consistency and projection migration challenges, as they have different complex issues to the addressed data conversion problem. Furthermore, this thesis does not explore diverging interpretations of Event Sourcing systems. The system built for the thesis is to be used for different data conversion methods for the sake of consistency. This thesis focus on the storing process to a database rather than database structures. A deployment strategy was decided upon. However, it was implemented in collaboration with the industrial supervisor using Docker containers and will therefore not be described as part of the implementation.
1.8 Contribution

The contributions of this study are summarized as:

- A reliable and available Event Sourcing proof-of-concept.
- Example of data conversions of attribute-driven events.
- An improvement in the operations table described by Overeem et al. [8].
- Example of a snapshot method that provides a solution to the drawback of long event streams in event stores.
- A motivation for choosing fine-grained events over coarse-grained events.
- Experimental results that support the claims of the validity of the implementation.

1.9 Disposition

Chapter 2 gives an extensive background on core concepts describing the Domain-Driven Design community, followed by a system design approach called Command query responsibility segregation. Chapter 3 describes Event Sourcing, explaining the different components of the concept and work related to this thesis. Chapter 4 describes the steps that were taken to realize proof-of-concept and how it was validated. Chapter 5 describes the chosen technology and definitions. Chapter 6 describes the core implementations for this thesis. Chapter 7 presents the results produced from the experiments. Chapter 8 presents an evaluation of the produced data and framework. Chapter 9 is a discussion of the evaluation, also possible alternative solutions. Chapter 10 presents the conclusion of this study and ends with a motivation for future work.
CHAPTER 1. INTRODUCTION
Chapter 2

Concepts and system design

This chapter encompasses the terminology, concepts and system design that sets the foundation for the thesis. Section 2.1 gives a brief description of core terminology and concepts stemming from the community of Domain Driven Design. The basics are then extended upon by a description on Command Query Responsibility Segregation system in section 2.2.

2.1 Domain Driven Design

There are core terminologies and concepts that Domain-driven Design and Event sourcing share. Domain-driven design is a concept for developing software with intent to solve complex needs by combining the implementation of an evolving model to core business concepts.

Concepts

Evans describes four essential concepts of Domain-driven design to concretize an otherwise abstract and complex system:

- **Domain** - A domain describes an area which the user applies a program which is the domain of the software.

- **Model** - Is an abstraction of a system that describes selected aspects of a domain. One application of the model is to use it to solve issues related to the domain.

- **Ubiquitous language** - The practice of building up a common, precise language between users and the developers. The language encapsulates the domain model.
• **Context** - Is the setting in which statements or words determine the meaning. Statements are only understood with context.

![Figure 2.1: Example of a domain model for a vehicle inspection program. Drawn by the author.](image)

The four concepts together form the **domain model**, which is a conceptual model of a domain. The model consolidates both behaviour and data of objects in the domain and modelled in a way that developer, domain experts and other involved parties may understand the program. Figure 2.1 shows an example of a domain model drawn for a vehicle inspection program.

**Building blocks**

The building blocks of a model-driven design abbreviate a foundation of best practice for object-oriented domain modelling [11, p. 70]. They guide decision making to clarify the model and to keep the model and implementation aligned, with one block bolstering the effectiveness of the others and vice versa. Although there are multiple building blocks, some of them serve no purpose for Event sourcing. The building blocks with relevance for this thesis are **entity**, **aggregate**, **service** and **value object**.
2.2. COMMAND QUERY RESPONSIBILITY SEGREGATION

An entity is an object, not defined by its attributes, but by identity and its thread of cohesion. An aggregate is an assemblage of objects viewed as a unit with the intent of data changes. A reference to an aggregate is restricted to a member of the aggregate assigned as root. A service is an operation presented as an interface which is independent in an abstract system, with no encapsulated state. A value object, in contrast to an entity, describes some characteristics or attributes and carries no concept of identity [12].

2.2 Command Query Responsibility Segregation

As mentioned in the introduction, the most common way to communicate with information systems is to use them as a CRUD data-store. As the need for complex systems has evolved, the way of tackling these issues has become more intricate [13]. CQRS presents a solution. However, the trade-off is an increase in system complexity, which cannot be avoided [14]. CQRS derives from a pattern called Command Query Separation, which introduced in 1988 by Meyers [15]. By applying the fundamentals of both Command Query Separation and elements from the Domain-driven design community, Young and Dahan [16, 17] presented CQRS.

The CQRS pattern describes two independent sub-systems, the command-side and the query-side, where command-side governs changes imposed by the users and query-side controls monitoring of states. The command-side is append-only, which renders the events into immutable occurrences, while the query-side may receive and reply to queries. Each sub-system has its data store, with the event store from the command-side projecting its data on to the data store on the query-side. The data store on the command-side projects its data to the database on the query-side through message busses or by polling [18]. Figure 2.2 describes the pattern.
An immense advantage of the CQRS pattern lies in the fact that each sub-system can have different hardware-and software-implementation. Any change or update of either implementation has no direct effect on the other system.

**Command-side**

The command-side is overall eminently similar to the stereotypical architecture of a relational database system, the contrast being behaviour instead of data-centric contracts. The command itself is a simple object containing various information depending on what type it is [16]. The Listing below 2.1 shows an example of a command object named Command in C#.

```
public class CommandObject{
    public Guid EntityId {get; set;}
    public object Command {get; set;}

    public CommandObject(Guid id, object command) {
        EntityId = id;
        Command = command;
    }
}
```

**Listing 2.1:** A command object in C# code.
When a command from the user interface is inquired, the command is applied to a designated aggregate or a new aggregate is created and stored in an event store. The actions are most commonly organized by a command handler that makes sure that commands are correctly stored \[17\]. The number of command handlers depends on the number of commands available in the user interface. Figure 2.3 shows a detailed illustration on the command-side.

![Command-side of the CQRS pattern](image1)

**Figure 2.3:** command-side of the CQRS pattern. *Drawn by the author.*

**Query-side**

The query-side only contains methods for retrieving data. As mentioned earlier, the query-side receives projections from the command-side, which can be stored in a traditional data store. The query-side, similar to the command-side, has a query handler, in charge of handling query data and returns data to the user. Figure 2.4 represents the query-side of the pattern.

![Query-side of the CQRS pattern](image2)

**Figure 2.4:** query-side of the CQRS pattern. *Drawn by the author.*

Having events sent and projected on query database is a powerful tool, however, there is a drawback. As the command-side receives a command and stores the command and events, the data-store on the query-side waits for an update for one of its projections. The query-side is usually eventually consistent.
CHAPTER 2. CONCEPTS AND SYSTEM DESIGN

2.2.1 CQRS and eventual consistency

In established teachings of databases, the four properties that are required are the ACID properties [19]. Another set of properties considered in distributed systems are Consistency, Availability and Partition tolerance. The CAP theorem also known as Brewer’s Theorem, suggests only two out of three can be fully satisfied for a distributed system [20]:

- **Consistency** - The condition states that all nodes see the same data at the same time. The read query is the most recent write query or an error message. Consistent systems start with a consistent state and end with a consistent state.

- **Availability** - Every request receives a response of success or failure. An available system is operational and responds at all times.

- **Partition tolerance** - The system runs, despite message delays caused by the network between the partitions. Partition-tolerant systems sustain any amount of network failures that will not result in failure of the entire network.

Distributed systems commonly use distributed databases. The most frequent needs for a distributed database is Availability and Partition-tolerance as consistency could eventually be reached [21]. A system implementing the CQRS pattern may provide an eventually consistent system, thus providing a reasonable solution.
Chapter 3

Event Sourcing system

The primary focus of this thesis is to evaluate an event sourced system and its complexity in relation to event granularity. Section 3.1 gives an introduction of Event Sourcing. Section 3.2 details effects in relation to event granularity in a simple event sourced system. Section 3.3 presents studies on Event Sourcing framework.

3.1 Event Sourcing

Event Sourcing derives from the Domain-Driven Design community. The motivation behind the concept of Event Sourcing is to have event-driven persist and atomic state changes. It provides complete audit logs of changes invoked on entities, leading up to the current state. The audit logs are sequences of events that explain the transformations an entity has gone through. In Event Sourcing an event has to be:

- An instance of past tense, the change had already been implemented and persisted before it was seen as an event.

- A description of the change that caused the state of an entity, e.g., an entity with title A has had a change which caused an event with the description: ”Title was changed to ’B’.”, the current title of the entity is now B.

- Immutable.

The events are stored as sequences of events related to one entity. A sequence of events in Event Sourcing is called an event stream and represents an aggregate portrayed in section 2.2 building blocks. Events are generated
CHAPTER 3. EVENT SOURCING SYSTEM

through commands that are sent to the command model, which are then persisted to the event store [4].

Figure 3.1: Append-only even stream. *Drawn by the author.*

The figure above shows an example of an append-only event stream, the large arrow indicate that the stream can only be read in one direction, while the smaller arrows express the link that connects the events to one stream and is represented by a unique ID. Figure 3.2 illustrates how a command prompts an application to create a new event to persist into an existing stream, located in the event store.

Figure 3.2: Simplified illustration of a command triggering an event *Drawn by the author.*

A drawback of Event Sourcing is an immense amount of reading operation invoked on the event store from the query-side. Whenever an entity that already exists in the system requires an update, to ensure correctness, it has to be replayed from the event store. A way to reduce the amount of reading operation is to extend the architecture with another database that maintains the current state for reading operations, which makes the CQRS pattern described in section 2.2 a fitting design.
3.1. EVENT SOURCING

The current state is only accessible through the event streams and the second drawback of such architecture can be a long event stream. An entire stream has to read from the event store. A common way to solve the issue is to use snapshots, which is a representation of the state of when it was captured and can be stored as an event. The stream can be played from the snapshot to the current state. In traditional snapshooting, for every N:th event a state snapshot is attempted, an alternative approach is rolling snapshots which project the current state by always attempting a snapshot for every new event pushed to the store [22]. Figure 3.3 illustrates a comparison of normal stream replay to a stream replay with traditional snapshooting.

3.1.1 Event store

The Event store is the storage used to deposit events. In contrast to a traditional database, an event store as an append-only data store, stored data can never be changed or deleted. A common representation of an Aggregate in the event store is with a table. The table contains information seen in Table 3.1:

- **Version** - The position of the event pushed to the stream.
- **Stream ID** - Indicates which event stream the event belongs to.
- **Event type** - The type of event that caused the change.
- **Created date** - When the event was created.
- **Data/event object** - Contains change data of the event.
CHAPTER 3. EVENT SOURCING SYSTEM

<table>
<thead>
<tr>
<th>Version</th>
<th>#ID</th>
<th>Event type</th>
<th>data</th>
<th>Created date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>@123</td>
<td>&quot;Asset created&quot;</td>
<td>{&quot;ID&quot;:&quot;@123&quot;}</td>
<td>2019-04-29 14:29</td>
</tr>
<tr>
<td>2</td>
<td>@123</td>
<td>&quot;Status changed&quot;</td>
<td>{&quot;Status&quot;:&quot;ok&quot;}</td>
<td>2019-04-29 14:29</td>
</tr>
<tr>
<td>3</td>
<td>@123</td>
<td>&quot;Title changed&quot;</td>
<td>{&quot;Title&quot;:&quot;Hello&quot;}</td>
<td>2019-04-29 14:29</td>
</tr>
</tbody>
</table>

Table 3.1: Visual example of an aggregate / event stream table representation.

The version column serves two purposes. The main purpose is to provide optimistic concurrency as a safeguard, that allows for multiple transactions to be completed without interfering with one another [23]. The secondary reason is for replaying events through snapshots. The stream continues from where the snapshot ended.

An event object can be represented by a JSON object [6][24] which is a collection of a key and value pair. The Event object contains vital information of the event such as the change data, stream ID, causation ID (an ID that tells what caused the event and the version of the event itself). An example of a **Title changed** event object is shown below in Listing 3.1.

```json
{
    "Title" : "Hello World!",
    "Aggregate ID" : "@123",
    "Causation ID" : "12345-12312",
    "Event version" : "1.0"
}
```

Listing 3.1: An example of a Title changed event, represented by a JSON object.

The Event object can be saved in a separate table with a unique identifier and referenced to in the data column, or the object could be saved directly to the column as a JSON string object. An Event Sourcing system can conceivably be implemented without CQRS. However, the Command Query Responsibility Segregation pattern works well with event-based models and the pattern provides a simple solution for reducing queries with the read model side.

A **reliable** event store as described by Overeem et al. [8] is database that does not undergo explicit change. The definition of **explicit change** is **direct modification** of the structure and/or information of events or event streams in the store.
3.2 Effects of varying granularity of events

As mentioned in section 3.1 an event has to describe its contents, but there is no clear definition of the number of attributes an event encapsulates. Although its attributes do not define an entity, changing its attributes provokes events. Ye [25] wrote a thesis that presents an evaluation of coarse-grained events and fine-grained events. Ye starts by defining events, clarifying and exemplifying their classifications:

Coarse-grained event - An event is big or coarse-grained if it contains information on more than one attribute of an entity, in other words, any command generates a single event no matter how complex. A single handler handles the event.

Fine-grained event - A small event or a fine-grained event depicts the information of a single attribute, a command object with multiple attribute changes generates multiple events.

The two classifications have similar domain logic, but require different handling. The purpose of this study was to find which type of event was most suited in a performance critical system. The metrics of interest were latency and storage usage. Latency measures the time it takes to store an event and size of storage measures the resulting size of the data store in bytes.

The results of the thesis showed favour in choosing coarse-grained events in terms of latency. However, the storage size was more than double compared to fine-grained event storage [25, p. 41]. The conclusion was that coarse-grained events was a viable approach and an optimal choice in terms of performance, however, Ye acknowledge bottleneck in the implementation for processing the fine-grained events.

3.3 Framework for handling data conversion

Managing data conversion is one of the most significant challenges of Event sourcing. A standard method of address complex issues is to divide the general problem into smaller real problems. In 2016 Spoor [26] presented a framework to handle data schema changes. The motivation behind the thesis was to help future developers by setting a starting point for building an Event Sourcing system. The conclusion suggests a solution by pinpointing and resolving four sub-challenges.

The first challenge was identifying operations that require schema changes and the difficulty of handling these operations (some operations found in table 3.2). The second challenge was to establish methods to handle the
operations and analyze their strengths and flaws. The third challenge to be resolved was to research deployment strategies to minimize downtime methods such as big flip, rolling upgrade and blue-green deployment. Lastly, a suggested framework was presented where all methods could be combined differently according to the needs of the domain. Spoor, later on, made a publication with Overeem et al. [8].

3.3.1 Evolving events techniques

A common occurrence in a traditional system is that data goes through multiple changes during their lifetime. In an Event Sourced system, events go through similar changes and they are called operations with a divergence of an operation being explicitly caused by the developer on certain parts of the systems and is not visible to the user. Operations hold different levels of complexity; basic operations are easy to handle and do not mutate the contents of an object substantially or at all. Complex operations require mutation, inspiring significant changes to the objects to content wise or structurally; they are harder to handle and require special techniques to be dealt with.

<table>
<thead>
<tr>
<th>Level</th>
<th>Complexity</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Basic</td>
<td>Add attribute</td>
<td>An attribute is added.</td>
</tr>
<tr>
<td>Event</td>
<td>Basic</td>
<td>Delete attribute</td>
<td>An attribute is deleted.</td>
</tr>
<tr>
<td>Event</td>
<td>Basic</td>
<td>Update attribute</td>
<td>An attribute is updated.</td>
</tr>
<tr>
<td>Event</td>
<td>Complex</td>
<td>Move attribute</td>
<td>An attribute is moved</td>
</tr>
<tr>
<td>Event</td>
<td>Complex</td>
<td>Split attribute</td>
<td>An attribute is split.</td>
</tr>
<tr>
<td>Event</td>
<td>Complex</td>
<td>Merge attributes</td>
<td>Attributes are merged.</td>
</tr>
<tr>
<td>Event-stream</td>
<td>Basic</td>
<td>Add event</td>
<td>An event is added.</td>
</tr>
<tr>
<td>Event-stream</td>
<td>Basic</td>
<td>Delete event</td>
<td>An event is deleted.</td>
</tr>
<tr>
<td>Event-stream</td>
<td>Basic</td>
<td>Rename event</td>
<td>An event is renamed.</td>
</tr>
<tr>
<td>Event-stream</td>
<td>Complex</td>
<td>Split event</td>
<td>An event is split.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Basic</td>
<td>Add stream</td>
<td>A stream is added.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Basic</td>
<td>Delete stream</td>
<td>A stream is deleted.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Basic</td>
<td>Rename stream</td>
<td>A stream is renamed.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Complex</td>
<td>Split stream</td>
<td>A stream is split.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Complex</td>
<td>Merge streams</td>
<td>streams are merged.</td>
</tr>
</tbody>
</table>

Table 3.2: A table of different operations. Table inspired by Overeem et al. [8].
The table above, Table 3.2 illustrates the different operations of each distinct level and their complexity. The process of conveying these operations holds no standard practice, they are often interpreted, the reaction is multiple techniques to handle these operations.

3.3.2 Techniques for handling complex operations

There are some already known approaches currently in use that handles the more complex operations. Each technique has its benefits and detriments that should be considered while implementing. The techniques listed below are the methods of interest for this thesis have been suggested by Overeem et al. [8].

Multiple versions - The technique is supported through the application layer. It does so by extending to a version number. All the event listeners are aware of every version of the events they are in charge of and has to be able to handle all of the events and their versions. In this technique, the event store is left as it is. Events are not converted and stored again. A downside to the technique is the increasing amount of duplicate code and code in general. Maintenance may cause significant downtime.

Upcasting - A technique that makes use of inheritance from object-oriented programming [27]. In Upcasting a new version of an event inherits from a predeceasing version of the same event. The conversion of events is controlled by an upcaster which holds information about the event types and event versions. The event listeners are not aware of event versions. The upcaster converts the events to the latest version so the listeners only have to support the latest version of an event. The technique is suggested and implemented in the Axon Framework [28] and by Betts et al. [29]. A downside to this technique is the increased workload on the domain side as the whole stream is checked for conversions, with multiple versions applied on top of one another, all in a serialized manner.

Lazy transformation - The Lazy transformation technique has similar logic to Upcasting. However, the converted event streams are also pushed to the Event Store. Sadalage and Fowler [30] further describe the technique. A benefit of pushing the converted stream in the reduced workload, streams no longer need full conversions. However, the Event Store is no longer considered immutable.
**Copy and transform** - A method utilized by the store and holds similarity to techniques described by Dumitras and Narasimhan [31]. The events in the old store are copied and transformed into the new store. The old event store stays intact and a new store is used instead.

### 3.3.3 Deployment strategies

An update technique is used to transform old representations of an event into a newer representation. However, when the system evolves with the events, a system upgrade often requires some downtime, something critical in any system. Deployment strategies can be applied to the system to address this issue. The figure below shows a system upgrade Figure [3.4](#).

![Figure 3.4: A system upgrade on the event store. Drawn by the author.](#)

The system upgrade also provides **system reliability** as the old events are not directly transformed into the old event store, but rather a new one.

The purpose of the deployment strategy is to execute a non-run-time upgrade technique. The deployment strategy is critical for the non-run-time upgrade as it requires the system not to run while the upgrade technique is being applied. **Multi-versioning**, **upcasting** and **Lazy transformations** are considered run-time upgrade strategies that are not meant to cause any change to the event store, while **copy and transform** is viewed as a non-run-time strategy. The deployment strategies considered for this thesis are the following:
3.3. FRAMEWORK FOR HANDLING DATA CONVERSION

**Big flip** - Described by Brewer [32], utilizes routing to manage traffic to one-half of the machines, while the other half is made available for upgrade. Once an upgrade is made, the traffic is rerouted to the upgraded half of the machines and so that the other half may upgrade. When all machines have been upgraded, the workload is re-balanced evenly between all the machines.

**Rolling upgrade** - The strategy also makes use of routing. The machines on this approach are upgraded in groups described by Dumitras et al. [33]. The main attraction of this strategy is the available machines to handle traffic at most times more than the machines that are being down for an upgrade, once the breaking point of more upgraded machines than non-upgraded machines, the traffic is swapped to the upgraded version. A downside to this method is that the machines have to run mixed versions of the application, which makes the approach complex.

**Blue-green** - The blue-green deployment described by Fowler [34], is a technique best described as a dual deployment method. Every application is always deployed twice, one with the current version of the application and one with either an older version or a future version. One of the deployments is set as the active while the second deployment is inactive, allowing the possibility of implementing upgrades on the inactive application, while the application is still running, eliminating downtime. The traffic is rerouted to the inactive application once the system is ready.
Chapter 4

Research method

This chapter incorporates specific steps of the scientific method. The chapter starts with a short summary of the steps in section 4.1 followed by section 4.2 describing the qualitative research. The theory is extended upon with section 4.3 that entails the description of the quantitative experiments.

4.1 Research steps

This research consisted of three research phases: A literature study, a practical study and an experimental phase that serves the purpose of validating and prove correctness. The literature study and the practical study were conducted following one another, while the experiments were conducted after a viable proof-of-concept was realized.

4.2 Qualitative research

4.2.1 Literature study

The literature study mainly consisted of research on Event Sourcing and Event Sourcing systems. The most relevant article for building an Event Sourced system, have already been presented in chapter 2 a paper written by Overeem et al. [8]. The paper suggested a framework for handling data conversion and provided some insight on what Event Sourcing was as a concept. The literature study was afterwards directed towards events [6 13 16 17 35] as a continuation study, to learn about what type of data was inserted into the system and if varying the content or method could affect the results.
The study of the query-side logic yielded a positive discovery. The logic of the query-side would be kept quite simple, similar to a traditional database [14, 16, 17]. Furthermore, a study on intelligent Agents [36] and possible machine learning and deep learning solutions [37, 38], however, both machine learning and deep learning remains as unexplored territory for Event Sourcing systems. The motivation for studying these topics was to gain insight on possible solutions for handling events from a new system version in an older system, the problem which was mentioned by Young [6], will be further elaborated in chapter [10] for future work. Lastly, the thesis was written by Ye [25] was studied to validate the experiment also justify the choice of the event type.

4.2.2 Practical study

The practical study began with revising prior implementation made by the company. The prior work consisted of a controller that received data and updates from users. A skeleton code logic that turns the command to an event and stores the data in an event store. As a result from the literature study, the command-side logic was understood to require heavier and more complex problem solving, a decision on defining framework and event type had to be made before the implementation could continue.

The new architecture inherited from the literature study as well as the core from prior work. When the final architecture and framework had been established, the implementation of the new architecture acting as a proof-of-concept was made. The process of implementation which was an iterative process and divided into two parts, implementing the command-side logic and implementing query-side logic, the ordering of implementation dictated by the command-side, as the query-side heavily relied on the command-side logic. The key implementations that were done for the command-side were the following: Data conversion and an improvement on store data with snapshots. The repository of the project can be found in the Appendix.

Store data

There were two use cases identified for storing data in the event store, a single write operation and a Read and write operation. The single write use case encompasses the action of creating a new entity, which is a write only operation, while the second use case is a read and write operation that covers the action of updating an already existing entity.
4.2.3 Validity

The validity of qualitative research is specified as the applicability of the assembled data for a given problem. The variable in question is reliability, affirmed by repeating the experience and produce the same or similar results. Such measurements are hard to prove, as qualitative research inherently is a subjective study. However, this thesis, as other research papers, is subjected to peer-review, which to some degree, provides validity.

4.3 Quantitative research

In regards to the related work written by Ye [25] and quantitative research conducted on this research, the validation of using coarse-grained events is in order. However, this thesis intends to validate fine-grained events as a standard choice. As mentioned, the use cases evaluated in this thesis are single write operation and Read and write operation. To confirm correctness of the quantitative research, the metrics measured are latency and complexity. The quantitative research for measuring latency was carried out as a set of experiments described and numbered below:

1. Experiment one - Inserting new events to an empty stream - The test case involves storing new events, to a clear stream, each case increases the number of events that are being stored to empty streams. The experiment tests the latency in response from the event store when the command handler invokes several changes to one entity; in other words, how long it takes to invoke a write operation. The experiment was carried out to evaluate use case Single write operation.

2. Experiment two - Inserting new event to different sized streams - The test case involves storing a new event to existing streams, the length of the stream increases each test case, the number of event stored remains as one. The experiment tests the latency of reads and writes and the experiment also serves as a way to validate the snapshot implementation. The purpose of the experiment was to produce data to be evaluated for the use case of Read and write operation.
To ensure that the system works properly, data conversion will be included and applied for all these tests and part of the in the latency measurements. Each test case of each experiment is executed 10,000 times to produce more accurate data.

Gauging complexity is done through an evaluation of table 3.2 described in section 3.3.1 in relation to the suggested framework. The metric of complexity also acts as catalyst to validate the chosen architecture and framework.
Chapter 5

Chosen technology

5.1 Programming language

The chosen language for this thesis is C# [39]. A useful language for object-oriented programming also has many features that make the implementation easier. C# was the language of choice by the company the proof-of-concept was implemented for.

5.2 Databases

Two database types were chosen for this research, one specializing in storing of events and the latter handles queries for complex data.

Event Store - For the database on the command-side, Event store was chosen [40]. Even store is a database optimized for storing of events, the main benefit of using event store is the reduced time spent on creating your event-driven database.

PostgreSQL - The database used for the query-side is PostgreSQL [41], an object-relational database system that allows for querying with complex data [42].

The secondary motivation for choosing these two technologies was because of the company the proof-of-concept was carried out for had these technologies already available to be used.
5.3 Fine-grained event

If we abide by the rules defining an event as described in section 3.1 and following the outline of section 3.2, a fine-grained event depicts the information of a single attribute. Having an event that describes attributes makes the event simple, with less information to change, which makes an attribute-driven event a suited representation.

```
{
  "Title": "Hello_world!",
  "Id": "1acfe483_a702_4d64_b640_d25d7bc4f041",
  "AssetId": "74acfd99_c298_400b_a83d_764bf8f5e8f0",
  "Type": "Domain.EntityTitleUpdated, Domain",
  "Version": "1.0",
  "Deleted": false
}
```

Listing 5.1: An event represented as a JSON object.

The listing above shows the core information of an event chosen for this thesis. The second line is the attribute title being changed, the third line is the ID of the object. Line four presents the event descriptor, it depicts what caused the event to happen, the following line displays the version of the event type. The fifth line is an indicator of the existence of the object. As mentioned before, events can never be removed, however, if the line is stated as true the object will not exist for on the query-side.

5.3.1 snapshot

The chosen method of snapshots is the traditional method; the motivation for choosing the traditional method is the reduced amount of times spent on accessing the snapshot stream with read and write operations.

Another motivation is that by using the rolling snapshot approach, it is more convenient to replace the fine-grained events as the snapshots are being stored more frequently. In theory, Rolling snapshot method is the same as coarse-grained events, but without a description of intent.
The listing above 5.2 shows an example of a snapshot, within the inner bracket the state of all the entities to a certain point in the stream are saved. The outside bracket contains critical information that ties the snapshot to the entity, where in the stream the snapshot was captured and lastly that it is a snapshot.

5.4 Postman

Postman [43] is a tool that enables testing of RESTful APIs [13]. The Postman API testing environment has an interface which makes HTTP request with either XML or JSON formatting and visualizes the response of the request, enabling easy testing and debugging.

5.4.1 Commands

For the narrative of this thesis, a command is a HTTP request that is being received by a RESTful API [14]. The commands that are handled in the proof-of-concept are, POST, PUT and GET, each command has its own handler:

- **POST** - Create a new instance of an object.
- **PUT** - Update attributes of the object.
- **GET** - Retrieve a stream of events.
The commands are to be handled by separate controllers, that invoke methods in a different pattern in order to store the events correctly. The motivation for choosing HTTP request is to emulate a real life request of a business environment.

```
{  
  "Title": "Hello_new_world!",  
  "Language": "English",  
  "Status": "OK"  
}
```

**Listing 5.3:** Example of a command.

A command received by the REST API is a JSON object, the content is the name of the attribute paired with the data. The listing above 5.3 shows an example of what a command changing three attributes looks like, the command itself is part of the command object sampled in listing 2.1.

5.5 Framework

The framework selected for this research is a product of the qualitative study. **Transform** (upcasting) and **Copy and transform** are the two data conversion methods chosen, while **blue-green** and **Big-flip** has been chosen for the deployment strategy. The framework and the operations will be further discussed in chapter 8. The goal of the chosen framework is to maintain **reliability** and keep the system **available**.
Chapter 6

Implementation

This chapter details the approach for implementing the proof-of-concept. The chapter starts with section 6.1 which describes the implementation of the command-side, followed by section 6.2 which details the query-side.

6.1 Command-side logic

The main logic of the command-side is the controller named command-Handler. Following the literature study, the core logic of the command-side adheres to the system overview shown below 6.1.

Figure 6.1: The system overview of the command-side. drawn by the author.
The commandHandler is responsible for receiving commands and store events. The three types of command handlers are the following: Create Entity Handler, Update Entity Handler and Get Stream Handler. The entity itself converts the command into attribute specific events that are added to a list of events. The command handler then stores events in the event store.

## 6.1.1 Create Entity Handler

The responsibility of create entity handler is to turn the command into events, create a snapshot of entity current state and store the newly added events in an event store.

```csharp
public class CreateEntityCommandHandler
{
    private IEventStore eventStore;

    public CreateEntityCommandHandler(IEventStore es)
    {
        eventStore = es;
    }

    public void ExecuteCommand(Command command)
    {
        var entity = new Entity(command.EntityId);
        entity.UpdateEntity(command.JsonObj);
        var listOfEvents = entity.GetListOfEvents(); 
        var snapshot = entity.CreateSnapshot();
        eventStore.AppendToStream(command.EntityId, entity.version, 
                                  snapshot.version, snapshot);
        entity.Version, snapshot.Version, listOfEvents, snapshot);
    }
}
```

**Listing 6.1:** CreateEntityCommandHandler.cs class C# code.

The listing above shows the CreateEntityCommandHandler class. UpdateEntity() and CreateSnapshot() are both methods from entity class.

UpdateEntity() methods responsibility is to convert our command object into a list of key-value pairs and create events according to keys. The object is converted using a library [45]. **Algorithm 1** shows a conversion of an object into a dictionary. The dictionary is iterated through and events are created and added to the listOfEvents accordingly.
Algorithm 1: UpdateEntity()

Require: listOfEvents as global

1: procedure UPDATEENTITY(command)
   
   eventDictionary = JsonConvert.DeserializeObjectDictionary<string, string>(JsonConvert.SerializeObject(command));

   foreach eventKey in eventDictionary:
   
      if eventKey == "Title" then:
         
         listOfEvents ← new EntityTitleUpdated(eventDictionary[eventKey]);
      
      else if eventKey == "Status" then:
         
         listOfEvents ← new EntityStatusUpdated(eventDictionary[eventKey]);
      
      else if ... etc

2: end procedure

The list named listOfEvents is the list that contains events of what has been invoked on the entity. A list is a key object that is being stored in the event store. CreateSnapshot() is a reversed update method. The snapshot method creates a dictionary that collects all the attributes in the entity as a key-value pair and converts it into a JsonObject with the library method JsonConvert.SerializeObject().

6.1.2 Update Entity handler

The update entity handler is responsible for updating attributes of an existing entity. UpdateEntityHandler has a similar composition as CreateEntityHandler with a slight divergent initial segment that require the previous state of the entity. Attaining previous state is done through requesting ReadStreamEvents() that returns a transformed stream of events and applied to the entity. Once all the events has been applied, the command is executed in a similar fashion. Listing 6.2 below shows the altered command executer used in UpdateEntityHandler.
```csharp
public void ExecuteCommand(object command) {
    List<Event> oldEvents =
        evenStore.ReadStreamEvents(command.EntityId);
    var entity = new Entity();
    entity = entity.ReplayOldEvents(oldEvents);
    entity.UpdateEntity(command.JsonObj);
    var listOfEvents = entity.GetListOfEvents();  
    var snapshot = entity.CreateSnapshot();
    eventStore.AppendToStream(command.EntityId, entity.version,  
                              snapshot.version, snapshot);  
    entity.Version, snapshot.Version, listOfEvents, snapshot);
}
```

Listing 6.2: Execute command in UpdateEntityHandler C# code.

`ReadStreamEvents()` requests from both snapshot stream and event stream and returns an event slice which is a list of old events. The maximum length of `listOfOldEvents` depends on how snapshot was implemented. If a snapshot is taken for every \(N\):th version, then the maximum length of `listOfOldEvents` from the event stream is \(N-1\), figure 6.2 illustrates expected length.

![Figure 6.2: Maximum length of listOfOldEvents. Drawn by the author.](image)

The third responsibility of `ReadStreamEvents()` is to request transform an event stream before returning it to the command handler, the transform algorithm is shown in subsection 6.1.4. `ReplayOldEvents()` applies the attributes of each event on to an entity by using a simple set method depending on what is found in the dictionary.
6.1.3 Get stream handler

GetStreamHandler receives a command which does not have a body. The command contains the Id of an entity that is requested for and applies the all the attributes to from the returned event slice and returns the entity. The execution command is much shorted for this command handler and the request looks similar to the starting lines from the previous handler. Listing 6.3 below shows the execution of GetStreamHandler.

1 public Entity ExecuteCommand(object command) {
2     List<Event> oldEvents =
3         evenStore.ReadStreamEvents(command.EntityId);
4     var entity = new Entity();
5     var event = entity.ReplayOldEvents(oldEvents);
6     return entity;
7 }

Listing 6.3: Execute command in UpdateEntityHandler C# code.

6.1.4 Data conversion with transformers

For handling evolving events the method that was chosen is a modified method of upcasting. The modified upcasting method targets the event type and its version and override the base Transform() method with an appropriate execution. There are two key algorithms used in this thesis for transformation, Algorithm 2 that receives the the list of JsonObjects form the event store and iterates through each event and request a transformation from Algorithm 3 responsible for checking if a transformation for an object is available or not and invoke the Transform() method.

Algorithm 2: transform algorithm

1: procedure TRANSFORMLIST(eventSlice)
   2:     listOfEvents = new listOfEvents;
   3:     foreach obj in eventSlice:
   4:         list ← TransformEvent(obj.data, obj.type, obj.version);
   5:     foreach eventObj in list:
   6:         event = JsonConvert(eventObj)
   7:         listOfEvents ← event
   8:     endforeach
   9:     endforeach
10:     return listOfEvents
11: end procedure
The second algorithm receives an event slice consisting of objects, not yet translated to events. The event slice is iterated through and the needed information is extracted from the object and used as parameters for the third algorithm, shown below Algorithm 3. Once all the events in the event slice have been iterated through, the objects are turned into events and returned to the command handler that made the request.

Algorithm 3: TransformEvent

1: procedure TransformEvent(oldObj, type, version)
   list = new list()
   transformerType = GetCustomAttribute(type, version)
   if transformationType == null then:
      return list ← oldObj
   else
      transformer = new EventTransformer(transformerType)
      tempList = transformer.Transform(oldObj)
      foreach obj in tempList:
         list ← TransformEvent(obj, obj.type, obj.version)
      endforeach
   return list
end

Algorithm 3 receives event data, type and version as parameters. A custom attribute is fetched by using GetCustomAttribute a C# system call. If a custom attribute does not exist, the untouched object is returned, if one does exist, an attribute transformer is created and the object is transformed. The full overview of the implementation is a typical stream processing implementation without using the processing tools. As mentioned before Transform() is a base class with classes extending the base class. The attribute determines which override method to use for the transformation. An example of how transformers were implemented for this thesis is in the appendix.
6.1.5 Event store

The event store implementation is responsible for receiving write and read requests. The three key methods implemented for this class are the following: `AppendToStream()`, `ReadStreamEvents()` and `ReadEvents()`. `AppendToStream()` was mentioned in subsection 6.1.1 listing 6.1 and subsection 6.1.2 listing 6.2. `ReadStreamEvents()` in subsection 6.1.2 listing 6.2 and subsection 6.1.3 listing 6.3. `ReadEvents()` is used by the Query-side.

Replay stream

`ReadStreamEvents()` is a method for handling read request by the command handlers. The parameter needed is the id of the requested stream, in return, the snapshot stream is read from the last saved entity snapshot. If a snapshot exists, the snapshot is added to the event slice and the event stream is read from when the snapshot was last saved, else the stream is read from start.

```
Algorithm 4: Read from snapshot stream and event stream

Require: [Access to an event stream and snapshot stream]

1: procedure ReadStreamEvents(id)
   eventSlice = new list()
   snapshot ← snapshotStream@id.getLast()
   version = 0
   if snapshot != null then
      version = snapshot.oldVersion
      eventSlice ← snapshot
   endif
   eventSlice ← eventStream@id.startFrom(version)
   return eventSlice

2: end procedure
```

The algorithm above Algorithm 4 illustrates how a `ReadStreamEvents()` procedure is done, the read from the event stream is an asynchronous call, which means that an exception is cast if the read fails and the exception is handled in the same fashion as the write operation, the whole process is executed again. The transformed events that were read from the event store are never treated as new events, an will therefore not be stored in the event-store, thus maintaining reliability.
Store events

`AppendToStream()` attempts to write to the event store. If a write was successful and the length of the event stream is longer than the threshold determined by the owner of the domain, a snapshot write is attempted, however, a successful snapshot store is not required. The algorithm below shows how `AppendToStream()` was implemented.

Algorithm 5: Store new events and attempt snapshots

Require: [Access to the Event Store, an event stream and snapshot stream, "Versions" acquired from replaying events.]

1: procedure APPENDToSTREAM(id, latestVersion, snapshotVersion, listOfEvents, snapshot)
   try from latestVersion:
     eventStream@id ← listOfEvents
   catch latestVersionWrongException:
     retry
     if (latestVersion - snapshot.oldVersion) >= threshold) then
       try from snapshotVersion:
       | snapshotStream@id ← snapshot
       catch Exception:
       | skip
     endif
   2: end procedure

The procedure is asynchronous, which means that if a write operation fails, an exception is cast. A common issue of writing to a shared space is a race condition, caused by multiple update request being made at the same time, `latestVersion` is a way to address this unwanted condition, in essence, optimistic concurrency. The specific exception shown in the algorithm above depicts when the position of the expected version does not match the current version, the exception causes chained re-attempted execution of the method in listing 6.2.

6.1.6 Copy and transform

The copy and transform method combines polling and the transform method presented by 6.1.5. The secondary event store polls from the currently running event store. The transform method is applied on all the events and written to secondary event store, one the polling method cannot get more...
events, a swap request is made if there were no changes to the current store during the request, the current event store gets swapped out by the secondary store.

6.2 Query-side logic

The logic of the query-side is a simple implementation. The query-side is only required to know of the current representation of the entity, the handling evolving schemas follow a deployment strategy **copy and transform** which is a non-run-time upgrade strategy; to solve that issue the **blue-green** strategy is combined.

The query-side database, as mentioned in chapter 5, is an object-relational database and the system relies on polling. There are three main implementations called `createTable()`, `AddEntity()` and `UpdateEntity()`. The polling method is used for the read model, where the controller of the database keeps requesting for updates from the event store.

6.2.1 Read model

The read model is the main controller of the query-side. The task of the read model is to request data from the event store and update the query-side database. The read model reads from the event store by calling the method `ReadEvents()` a procedure that requests for an event slice from the version it last requested and received a new event slice. The procedure below illustrates the implementation of `ReadEvents()`.

**Algorithm 6:** Request event slice from version

<table>
<thead>
<tr>
<th>Require:</th>
<th>[Access to an event stream]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure <strong>READEVENT</strong>(id, version)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>eventSlice = new list()</code></td>
</tr>
<tr>
<td></td>
<td><code>eventSlice ← eventStream@id.startFrom(version)</code></td>
</tr>
<tr>
<td></td>
<td><code>return eventSlice</code></td>
</tr>
<tr>
<td>2: end procedure</td>
<td></td>
</tr>
</tbody>
</table>

Since the read model keeps up with the event store by polling and keeps track of which version it has reached currently, snapshots are not required. If the event slice contains events (check the length of the list), convert the list into a dictionary and execute both `AddEntity()` and `UpdateEntity()`. Some other used database methods such as `getEntity()` a get request and `deleteEntity()` a delete command are also found in the query side.
Create table

The create table method is called each time the system each time the read model starts. The schema applied for the table is always set on the current representation of the attributes, a method best done manually. To communicate with the database, we use queries to send commands. The query for creating a table can be seen in listing 6.4 below.

```sql
"CREATE TABLE IF NOT EXISTS entities ( " + "id VARCHAR(80) PRIMARY KEY," + " title VARCHAR(255)," + " status VARCHAR(80)," + " language VARCHAR(80)," + " createdBy VARCHAR(80)," + " deleted VARCHAR(10)" + ");"
```

Listing 6.4: Create table query sent to database C# code.

The query command creates a table with the column keys set as id, title, status, language, createdBy and deleted, all of which represents an attribute of an entity. The keys are set to identify which column it belongs to, when an update command is made, the key is declared in the command to change the data.

Add entity

The add method is an automatic process invoked by the polling method. AddEntity() is invoked when an event with the type EntityCreated was found at the start of the event slice the read model receives. The query command for add is insert, the listing below shows an insert query.

```sql
"INSERT INTO entities (id ) VALUES (" + event.EntityId + ");"
```

Listing 6.5: A create query sent to database C# code.

Entities is the table that the entity is intended to be added to. The value is the identifier of the entity. The command is executed and the entity is added to the database and displayed as a new row on the database table.
**6.2. QUERY-SIDE LOGIC**

**Update entity**

The Update entity method receives a key-value pair from the read model, excluding the event mentioned above. The parameters for `UpdateEntity()` are `dictionary` and `entityId`. The dictionary is looped for each item it contains and executes the command below in listing 6.6.

```
"UPDATE entities" + "SET " + key + "=" + dictionary[key] + " WHERE id=" + entityId + " ;
```

**Listing 6.6:** An update query sent database C# code.

Although the store is able to handle objects, all of contents in this proof-of-concept can be represented as strings and handled accordingly within the model, the query-side looks very similar to a traditional database implementation, with the exception of it being a secondary database that reads from another data storage.

**Get entity**

Get consists of two methods, a `GetAll()` request without specifying which entity to get and `Get()` request which receives a parameter. The `Get()` request method returns an entity with all the attributes paired to one another, while the `GetAll()` request returns a list of all the existing entities.

```
"SELECT * FROM assets ;
```

**Listing 6.7:** no parameter get request database C# code.

```
"SELECT * FROM assets WHERE id=" + id + " ;
```

**Listing 6.8:** get single database C# code.

Listing 6.7 shows the `GetAll()` query command, listing 6.8 shows the single entity `Get()` query command. A get method receives all the attributes of an entity and applied on to an object.
Delete entity

If an entity to be removed from the query-side database, a special event must be found at the end of an event slice, an event called \textit{EntityDeleted}. The read model then executes a remove command as seen below.

\begin{verbatim}
"DELETE FROM assets " + "WHERE " + "id='" + id + "'
\end{verbatim}

\textbf{Listing 6.9:} query sent database C# code.

The delete method executes the query command above, which removes the entity from the query-side database. Note that the events of the entity still remains in the event store.
Chapter 7

Results

In this chapter a short summary of the implementation is presented followed by the results from the experiments. The chapter starts with a short summary of the implementation in section 7.1 followed by section 7.2 presents the execution and results from both experiments.

7.1 Implementation summary

The command-side command handlers provide a simple but effective solution that can easily be paralleled when it comes to handling the conversions. The transform method has a similar approach as a stream processing method and leaves the basic and complex operations to be handled within the command-side domain.

The query-side resulted in a simple read model that uses polling to read from the event store and update the object-relational database by executing query methods on newly polled events.

7.2 Experimental results

The experiments are assessments on the response time of the command handlers: CreateEntityHandler 6.1.1 and UpdateEntityHandler 6.1.2. The results of experiment one 7.2.1 are the outcome that was produced by CreateEntityHandler. The results of experiment two 7.2.2 is a product of the UpdateEntityHandler.
CHAPTER 7. RESULTS

7.2.1 Experiment one - Create new entity

The goal of the first experiment was to examine the time it takes to store events. There are 10 test case that evaluates the latency of storing different sized payloads of events in one single write request and the amount of storage they use. The table 7.1 below shows a summary of the first experiment.

<table>
<thead>
<tr>
<th>test case</th>
<th>number of events</th>
<th>Latency with standard deviation($\sigma$) (milliseconds)</th>
<th>storage size after write (kilo byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.0261 ±0.3739 ms</td>
<td>0.8 kb</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0.1596 ±1.8265 ms</td>
<td>8.1 kb</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>0.8171 ±6.4702 ms</td>
<td>81 kb</td>
</tr>
<tr>
<td>4</td>
<td>1,000</td>
<td>6.2425 ±17.7859 ms</td>
<td>811 kb</td>
</tr>
<tr>
<td>5</td>
<td>5,000</td>
<td>32.6597 ±16.0320 ms</td>
<td>4011 kb</td>
</tr>
<tr>
<td>6</td>
<td>10,000</td>
<td>57.1443 ±20.7685 ms</td>
<td>8107 kb</td>
</tr>
<tr>
<td>7</td>
<td>30,000</td>
<td>245.4489 ±102.8950 ms</td>
<td>$\approx$24000 kb</td>
</tr>
<tr>
<td>8</td>
<td>40,000</td>
<td>335.0011 ±130.5448 ms</td>
<td>$\approx$32000 kb</td>
</tr>
<tr>
<td>9</td>
<td>50,000</td>
<td>431.0565 ±179.6058 ms</td>
<td>$\approx$41000 kb</td>
</tr>
<tr>
<td>10</td>
<td>100,000</td>
<td>675.4853 ±202.1954 ms</td>
<td>$\approx$81000 kb</td>
</tr>
</tbody>
</table>

Table 7.1: Latency and storage size for experiment one.

The experiment encapsulates the time spent on a write operation to the event-store; each test case increases the payload of one single write request. The column number of events highlights the payload size. The column latency shows the mean and standard deviation of 10,000 program executions for each test case. The results were approximated with four decimals and the latency measured in milliseconds. Column storage size showcases the size of the events in kilo bytes once they had been stored, with test case 7-10 estimated.
7.2. EXPERIMENTAL RESULTS

7.2.2 Experiment two - Update entity

The goal of the second experiment was to measure the latency of both read and write operation, i.e an update and the storage space used. Similar to experiment one, 10 test cases were set up for the experiment where each test case has an increasing read-load of events in the event stream. The table below Table 7.2 shows a summary of the second experiment.

<table>
<thead>
<tr>
<th>test case</th>
<th>Size of event slice</th>
<th>Latency with standard deviation($\sigma$) (milliseconds)</th>
<th>storage size after write (kilo byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3.3451 ±2.5563 ms</td>
<td>0.8 kb</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>11.4483 ±5.7190 ms</td>
<td>8.1 kb</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>108.2491 ±17.5349 ms</td>
<td>81 kb</td>
</tr>
<tr>
<td>4</td>
<td>1,000</td>
<td>905.8261 ±120.3058 ms</td>
<td>812 kb</td>
</tr>
<tr>
<td>5</td>
<td>3,999</td>
<td>1998.3728 ±78.1302 ms</td>
<td>3254 kb</td>
</tr>
<tr>
<td>6</td>
<td>4,000</td>
<td>2.8074 ±3.3085 ms</td>
<td>3256 kb</td>
</tr>
<tr>
<td>7</td>
<td>5,000</td>
<td>537.6174 ±75.4171 ms</td>
<td>4022 kb</td>
</tr>
<tr>
<td>8</td>
<td>40,000</td>
<td>4.6639 ±4.8182 ms</td>
<td>$\approx$32000 kb</td>
</tr>
<tr>
<td>9</td>
<td>50,000</td>
<td>1956.3440 ±815.5370 ms</td>
<td>$\approx$41000 kb</td>
</tr>
<tr>
<td>10</td>
<td>100,000</td>
<td>4.3494 ±4.2264 ms</td>
<td>$\approx$81000 kb</td>
</tr>
</tbody>
</table>

Table 7.2: Latency and storage size for experiment two.

The second column represents the number of events that exist in an event stream in the event store, the size of event streams was tailored to accommodate the snapshot threshold that says; for every 4,000 events, a snapshot is taken. The third column presents latency of reading the stream and writing a single event to the store. The latency is the mean and standard deviation of 10,000 program runs for each test case, approximated with four decimals in milliseconds. The fourth column presents the storage size of the event store in kilo bytes, test case 8-10 are approximations.
Visual representation of both experiments

The block diagram and graph in figure 7.1 illustrates the results of the first experiment. The thin dark vertical lines portray the standard deviation. The second figure, figure 7.2 exhibits a block diagram and a graph of the produced results for experiment two, dark lines represent the standard deviation of the results.

Figure 7.1: Results latency of test cases for CreateEntityHandler plotted as a block diagram and a graph.

Figure 7.2: Results latency of test cases for UpdateEntityHandler, plotted as a block diagram and a graph.

7.2.3 comments

As stated in delimitation, the point of interest lies in latency and not in storage size. However, it is worth mentioning that the slight increase in storage usage in the second experiment compared to the first experiment, is due to the snapshots stored.
Chapter 8

System evaluation

The system goals of the proof-of-concept were for it to handle the operations that were introduced in chapter 3.3.2, however, further studies resulted in the removal of a few operations.

<table>
<thead>
<tr>
<th>Level</th>
<th>Complexity</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Basic</td>
<td>Add attribute</td>
<td>An attribute is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete attribute</td>
<td>An attribute is deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Update attribute</td>
<td>An attribute is updated.</td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>Split attribute</td>
<td>An attribute is split.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Merge attributes</td>
<td>Attributes are merged.</td>
</tr>
<tr>
<td>Event-stream</td>
<td>Basic</td>
<td>Add event</td>
<td>An event is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete event</td>
<td>An event is deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename event</td>
<td>An event is renamed.</td>
</tr>
<tr>
<td>Event-store</td>
<td>Basic</td>
<td>Add stream</td>
<td>A stream is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delete stream</td>
<td>A store is deleted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename stream</td>
<td>A stream is renamed.</td>
</tr>
</tbody>
</table>

Table 8.1: Operations handled by proof-of-concept.

The table above shows the operations that remained in the system, Table 8.1. The choice of removing some operation was due to the lacking necessity to keep them. On the event-stream level, the complex operations had technically not been removed, but rather, merged to the event level. Splitting/merging an event with the only changeable content, an attribute, the operations of split event and merge events holds equal purpose as splitting and merging attributes.
The complex operations for the event store were removed from the table, as the use case could not be justified in any report. By splitting or merging streams, it is hard to justify reliability, thus the removal of such operations. However, the deployment strategy remained the same, which makes adding the complex store operations a simple task. The delete operations remained, however, events and streams are never removed from the system, but rather ignored.

![Diagram](image)

**Figure 8.1:** Command-side framework inspired by Overeem et al. Drawn by the author.

As the operations were altered, the chosen framework described by figure 8.1 reflects the handling of operations well and covers the requirement of a reliability and availability system, with the exception of the system crashing due to unforeseen events.

The system works well with multiple versions of each attribute. However, the transformation method contributes to the latency, since the whole event slice has to be iterated, a method which has been optimized.
Chapter 9
Discussion

In this chapter, the results and the validity chosen methods and implementation are discussed. The chapter begins with an analysis of the results produced by both experiments 9.1, followed by section 9.2 discussing the validity of the implement and the chosen methods. The chapter ends with a critique of the study.

9.1 Results Analysis

During the literature study and experimentation, the biggest flaw of using fine-grained events became more apparent. The number of events grows at a high rate, the bottleneck lies with the read and write operations of the system, also known as an entity update.

The results from the first experiment show a linear trend with a growing standard deviation for each test case, which expresses how much the samples differ from the mean value. Although a write operation with 100,000 might not take place in a real-life situation, the results show that the request causes some level of stress on the database inducing some writes to take longer, thus a higher level of deviation. A more common write case would contain events ranging between 1 and 5,000.

The second experiment shows promising results except for the second half of the test cases, where the values fluctuate more, causing a higher standard deviation. The outcome could be directly linked to the issues from the first experiment, since setting up of the test case involved a write operation with large amounts of events, while still being processed by the database, a new read and write operation then provoked some level of latency. That is to say, the experiment was a bit rushed.
The biggest flaw of using fine-grained event becomes obvious for the second experiment, which is the number of events stored in a stream which lead to longer reads. A secondary reason is the amount of read and write operations invoked by the handler. Typical handling of an update command requires in total three read/write operations, one read from the snapshot stream, one read from the event stream and one write to the event stream. Once every 4,000 events, there is an additional write operation, done to the snapshots stream, which totals in four read/write operations.

The observation of these results is that a well implemented snapshooting method is essential as it shows that it can reduce the overall time spent on reading. Each decline in the graph and block diagram seen after the test case with 3,999 events indicates that there was a snapshot saved from the previous state change, causing the read operation for the event slice to contain fewer events. The best case for the read and write operation, was an event slice containing only one event or a snapshot. The worst case for the second experiment was an event slice containing 3,999 events.

In terms of correctness for the produced data of both experiments, the quantity of 10,000 program executions for each test case should advocate for a higher level of accuracy of the measurements.

9.2 Validity

The validity of the implementation can be seen in both reliability and the experiments. The events in the event store are never directly transformed, the copy-and-transform method only copies the stream to a new stream and leaves the old stream intact. The responsibility of keeping the correct transformations lies in the domain of the command-side, which makes the system easier to handle and develop. The results from the experiments enhance the validity of the system, a common case would be a write operation with a range between 1 to 5,000 events to be updated.

Given any of the cases presented in the first experiment, the response time is still well under 1 second, a response time which is common in a real-life system. The read operation suffers for the worst case when the event slice reaches a maximum length of 4000 events with the worse sample taking over 3 seconds to complete. The events also require transformation by the transformer according to algorithm 2 and algorithm 3, a process that contributes to the latency, but also implemented well enough for the list to only be iterated once.
To maintain a sustainable update handler, a suggestion to only have at most one version of an attribute transformation. Any more than that, the copy-and-transform method should be applied to the system.

The pros of using fine-grained events are quite evident, the intent of the event encapsulates the essence of its contents, also mentioned by Ye [25]. Other benefits that have been shown during the implementation is the simplicity of the data conversion model, as well as the handling of the data, where the data can be extracted rather quickly. Fine-grained events avoid unnecessary operations attribute extraction of events. The Data conversion methods for coarse-grained events would require more complex implementations as the intent has a broader scope.

One of the significant challenges faced during the implementation was the attribute merge operation. The solution that was found was to keep a set of past events that uses event-type as key and events as value, with the events to the transformer. The transformer applies the transformation and saves it in the set, which helps when two attributes require merging.

Rolling snapshots require more read and write operations than traditional snapshots, the sacrifice of lesser than 4 seconds to reduce the workload on the event store is valid, as read and write operations have been shown to cause stress on the system.

In terms of validity of complexity, which inherently is hard to prove in an objective matter, following the tables described by Overeem et al. [8] and the table in chapter 8 table 8.1 there is, in fact, a less complicated operation handling on the event-stream level, whilst the event-store level is defended as a subjective choice.

The system follows the CQRS pattern, as we allow eventual consistency measuring latency for when the query-side database reaches the same state holds no interests.

### 9.3 Critique

Two experiment that this thesis could have benefited from was snapshot optimization and read without snapshots. snapshot optimization an experiment where snapshots thresholds are tested to find an acceptable worst case latency. Read without snapshot would strengthen the validity of the implementation, by showcasing the amount of time spent on reading and transforming huge amounts of events, something that might have been the bottleneck found in the implementation by Ye [25].
Chapter 10

Conclusion

The goal of this study was to evaluate Event Sourcing and provide a proof-of-concept using fine-grained events and explore how the choice of events might affect the implementation of the system.

The study resulted in a working proof-of-concept, that handles data conversions by using a transform method. The proof-of-concept also upholds the goal of preserving both required attributes reliability and availability.

By using fine-grained events, it was also found that some complex operations were no longer in need, which reduced system complexity drastically. The bottleneck of the fine-grained events was also addressed by adequately implementing a system following the suggested framework in chapter 8 figure 8.1 with a well thought out snapshots strategy as seen in the command-side implementation section 6.1.

Future work

Further study to improve snapshots could be made. As stated in the discussion, to transform a list of events induces a fair amount of latency to the read processing, by finding benchmarking a good snapshot threshold with consideration to the trade-offs would be beneficial to an Event Sourcing system using fine-grained events. Stream processing has become more common and an approach of data conversions using stream processing could result in an interesting system.
Another interesting approach for handling data conversions would be the use of Intelligent Agents, who are tasked with the job of handling the data conversions. An interesting continuation study would be the mining of events, what type of information can we generate from mining changes in a database rather than the current state?
Lastly, one major flaw in Event Sourcing that lacks study, is the handling of events in a system rollback, multiple data conversions lack support in an old system and result in vast amounts of lost data.
Bibliography


[Accessed online, May 25 2019].
APPENDIX A

Full repository: https://github.com/HenkeNg/Master-thesis-project