Marcus Blikra Akre

Event log pruning in CQRS systems

Master's thesis in Applied Computer Science
Supervisor: Rune Hjelsvold
December 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Marcus Blikra Akre

Event log pruning in CQRS systems

Master’s thesis in Applied Computer Science
Supervisor: Rune Hjelsvold
December 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Abstract In CQRS+ES (CQRS with event sourcing) systems, we have separate data models for reading from and writing to the application. Both these models are based on the same set of events as a single source of truth. This means that every change to the application is stored as a new event. E.g., changing a person's home address or adding an order line to an order would be distinct events. By reading back all events from when the application was initially launched, we can reconstruct the current state of the application. This state hydration process becomes harder to manage over time, as more and more events are added to the event log. A common way to handle this problem is to use snapshots. By looking at state hydration in tandem with event transformation (log pruning), and persistence mechanisms, it is possible to take a holistic view of the command side of CQRS+ES architectures. The goal is to provide developers with a more diverse toolset when designing CQRS+ES systems.

We developed a CQRS+ES system with support for both event sourcing and command sourcing. In addition, we implemented multiple log reduction (pruning) and persistence mechanisms. By running simulations on this system, we were able to test and measure the performance of various CQRS+ES configurations. By doing this we gained insight into the design principles and performance of CQRS+ES systems. We were also able to provide alternatives for hydrating current state through command sourcing and pruning algorithms.

Keywords: CQRS, event sourcing, command sourcing, log pruning, state hydration, and event store.

Contents

1 Introduction **6**

1.1 Problem description 6

1.2 Motivation 6

1.3 Research question 7

1.4 Planned contribution 7

1.5 Outline 8

2 Background **9**

2.1 Event-sourced systems and CQRS 9

2.2 Aggregates 10

2.3 Event store 10

2.4 Events 10

2.5 Event streams 12

2.6 State 12

2.7 Projections 13

2.8 Commands 13

2.9 Sourcing type 14

2.10 Snapshots 14

2.11 Pruning algorithms 14

2.12 Pruning execution 16

2.13 Storage 17

2.14 Summary 17

3 Related work **18**

3.1 Event storage 18
3.2 State hydration 19
3.3 Log pruning 20
3.4 Summary 21

4 Methodology **22**

4.1 Simulation case 22
4.1.1 Model parameters and data generation 23
4.1.2 Model interaction 24
4.2 System design 25
4.2.1 Architectual overview 25
4.2.2 Aggregates 25
4.2.3 Commands and events 26
4.2.4 Command dispatcher 27
4.2.5 Streams and state hydration 28
4.2.6 Event store 29
4.2.7 Persistence 29
4.2.8 Sets and projections 32
4.2.9 Pruning process 32
4.2.10 Pruning algorithms 33
4.3 Simulation process 34
4.3.1 Process 34
4.3.2 Parameter sensitivity 36
4.3.3 Measurements 36
4.3.4 Software stack and hardware 36
4.4 Summary 37

5 Results **38**

5.1 Command execution 38

5.1.1 Data 38

5.1.2 Results 39

5.2 Event log pruning 39

5.2.1 Data 39

5.2.2 Results 40

5.3 State hydration 41

5.3.1 Data 41

5.3.2 Results 42

5.4 Parameter sensitivity 44

5.4.1 Data 44

5.4.2 Results 44

5.5 System implementation 46

5.6 Summary 48

6 Discussion **49**

6.1 Event log pruning and state hydration (RQ 1.1) 49

6.1.1 Event log pruning 50

6.1.2 State hydration 50

6.2 Command sourcing and state hydration (RQ 1.2) 52

6.3 Event granularity (RQ 1.3) 54

6.4 Design considerations (RQ 1.4) 55

6.4.1 A command-sourced CQRS system 55

6.4.2 A CQRS system with pruning 56

6.4.3 Projections and command execution 57

6.4.4 Implementing event stores 58

6.5 Reproducibility 59

6.6 Limitations 59

7 Conclusion 61

7.1 Future work 62

Appendices 68

A Running experiments 68

B Simulation parameters 72

C Test results 74

1 Introduction

This thesis will look into the area of event log management for software systems built using CQRS (Command Query Responsibility Segregation) [26] with events as a primary source of application data and state. The construction of the log handling mechanisms and the application data model must be done so that it is possible to estimate future performance as the event log grows. To enable his, we can apply retroactive pruning mechanisms, event log branching, and various persistence mechanisms. This will provide developers with a more diverse toolset when constructing CQRS applications than what is available in current literature.

1.1 Problem description

In CQRS+ES (CQRS with event sourcing [11]), we have separate data models for reading from and writing to the application. Both these models are based on the same set of events as a single source of truth. Every change to the application state is then stored as a new event. By reading back all events from when the application was initially launched, we can reconstruct the current state of the application. This must be done every time the application receives a request to modify state. This hydration process becomes harder to manage over time as more and more events are added to the event log. Event log granularity can also affect this problem as one event can be represented as many small smaller events, or as one large event.

1.2 Motivation

CQRS is a relatively new software architecture pattern (coined by Greg Young [26]). Much of existing literature on event sourcing uses snapshots (pre-processed state from past events) to solve the problem of having too many events [26][11][8][20][6]. A different approach is to increase performance by distributing data over multiple nodes [4]. Snapshotting is an effective solution, but it adds complexity and timing issues related to building and re-building the snapshots. At some point snapshotting might not be viable, and thus it should be compared with alternative strategies.

For developers, there is currently little research available on the topic of event sourcing performance. In [25] Brian Ye tests coarse-grained events based on existing finer-grained events. In [7] Benjamin Erb et al. takes a similar but more refined approach by applying different strategies on which events should be transformed into coarser-grained events. Still, there is more to explore in this area. By looking at event transformation in tandem with storage, and retrieval mechanisms it is possible to take a holistic view of CQRS+ES architectures, enabling the developer to make the right choice for their specific use case.

1.3 Research question

When looking at related literature, we can see that some of the issues of log pruning and query performance have been addressed before: Event granularity [25], log pruning [7], event transformation [19] and evaluation of event store performance [20][12]. However, these issues have not been tested together to investigate their impact on state hydration, and CQRS+ES system design.

By surveying existing literature, it is possible to define a selection of relevant log pruning mechanisms. These can differ on event granularity, sourcing types (event sourcing/command sourcing), and pruning approaches (bounded/probabilistic). Implementing pruning mechanisms, a workload generator, and event stores for different database management systems will enable us to test the implications of various pruning and persistence configurations on CQRS system designs.

This leads us to the main research question:

1. How can pruning and persistence of events affect state hydration and design of CQRS+ES systems?

To help address the main research question we have defined multiple subquestions:

1.1 Which effect can pruning algorithms have on hydration performance?

1.2 Command sourcing can be classified as a form of log pruning. What is its effect on hydration performance?

1.3 To what extent does event granularity affect hydration performance and event retrieval?

1.4 Which design considerations related to state-building and persistence must be considered when implementing a CQRS+ES system?

By answering the research questions, we address the research problem by providing guidance towards which pruning and persistence mechanisms that can be used to avoid performance problems for various CQRS+ES use cases.

1.4 Planned contribution

By investigating hydration performance, we will build on previous works identifying the need for more work on performance measurements on different database types [20][25], and on different domain models [25]. The thesis will provide insight into the performance of pruning algorithms combined with different storage solutions. The end result will be to provide alternatives to snapshots for

hydrating state in CQRS+ES within acceptable performance intervals.

1.5 Outline

We set out to present relevant CQRS+ES theory in section 2. Section 3 contains an overview of existing works on event storage, state hydration, and log pruning. Section 4 presents our research approach, including defining, implementing, and running simulations on a vessel fleet management system. Findings from the implementation and simulations are presented in section 5 and discussed in section 6. The conclusion is presented in section 7. Details on how to download and run the vessel management system tests can be found in appendix A. In appendix B we have documented simulation parameters. In appendix C we have included instructions on how to obtain measurement data, including test parameters for each simulation.

2 Background

In this section, we provide an overview of the CQRS (Command Query Responsibility Segregation) architecture. We also describe relevant concepts for working with events, and event streams and the implications these concepts have on the design space of CQRS systems. The main focus is on the write side of CQRS as this is the part relevant to the research question, but we will also include some elements from the read side.

In this paper, we use a vessel management system to run simulations and drive discussions. It is also used in this section for illustrative purposes. This system is further described in section 4.1. The core of the vessel management system is to manage a fleet of vessels and their work timeline (a set of contracts, called fixtures, and vessel positions).

2.1 Event-sourced systems and CQRS

Event-sourced systems use events as their source of application state instead of using direct state manipulation. When events are emitted, they are added to an event stream. Event handlers can then walk through the streams, and construct state gradually. CQRS [26] utilizes this concept but divides the system into two parts: Command, and query. We call these systems CQRS+ES. The command side is responsible for processing commands and emitting events. To be able to execute commands, current state must be calculated. This enables decision making. The query side uses the same events as a source to build various representations of the same events. E.g., data structures for user interfaces, reports, and command input. These data structures are called projections.

Figure 1 shows how commands are executed in CQRS+ES systems. An incoming command causes the system to fetch relevant events (events 1, 2, and 3). These events are processed in event handlers on aggregates to calculate current state. The command is executed, and a new event is emitted and saved. This new event is processed (along with events 1, 2, and 3) later on when a new command is received. In this example, we have two states. One current state for the execution of the command (no periods in the vessel timeline), and a future current state, not currently used (one period in vessel timeline).

CQRS is based on the more general CQS (Command Query Separation) principle. This principle states that the handling of commands (modifying state) and queries (reading state) should be separated. "Asking a question should not change the answer" [16].

Layer	Example operations
Store	Add, delete, rename, split and merge stream
Stream	Add, delete, rename, split and merge event. Move attribute to other event type.
Event	Add, delete, update, merge, split attribute.

Table 1: Event store layers and operations [19]

2.2 Aggregates

Domain data can be organized in aggregates. An aggregate contains a root that controls access to and operations on data in the aggregate. The aggregate forms a consistency boundary. Using aggregates as a partitioning construct in CQRS+ES reduces the number of events to be processed for each command. Thus, commands are only allowed to modify one aggregate [25][26][4][3][20]. In cases where multiple aggregates need to be modified, process managers (sagas) can be used to orchestrate a process.

2.3 Event store

The event store sits in the core of a CQRS system and is responsible for managing events, streams of events, and operations on these data structures. See upper part of fig. 1. An event store is often divided into three layers [25][19], with its corresponding operations. In the top layer we find the store itself that has operations for working on streams (e.g., adding, and splitting streams). On the middle layer, we find the steams. This layer has operations for working on events (e.g., query stream, add, split and delete events). Each stream can be defined with varying scope. E.g., one stream for the whole system or one stream per aggregate or set of aggregates. Each stream contains all events for the given scope. This way current state can be reconstructed. On the lowest layer, we find the events. This level has operations for working on event attributes (e.g., add, merge and delete attribute). By combining basic operations on the store, stream, or event layers it is possible to do more complex operations.

2.4 Events

Events are the core concept of a CQRS+ES system. Each event represents some delta of state change. E.g., "vessel-crated" or "position-reported". Accompanying each event is also an event payload (application or business-oriented event data). The payload varies depending on the type of event emitted. E.g., for "position-reported" we would also need to know to which vessel it belongs to make any sense of the event. By combining all events from the first to the last, it is possible to construct current state.

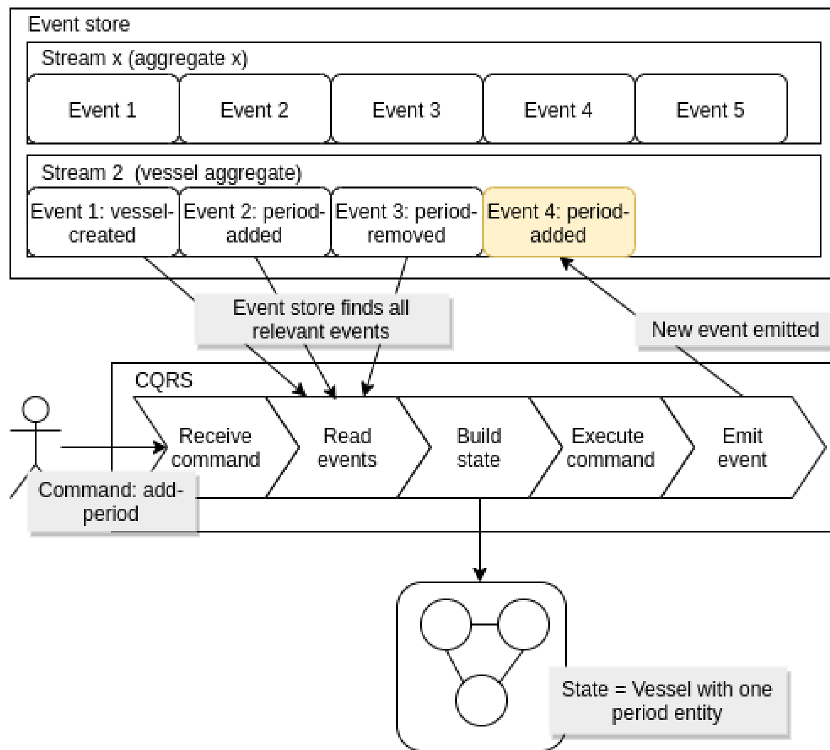


Figure 1: Receiving commands, building state and emitting events: Event 1, 2 and 3 is used to build state for decision making. The command produces a new event 4 during execution.

Much of the literature on CQRS+ES (e.g., [26][25][20]) suggests using a simple data structure for storing events. This structure consists mainly of an id identifying a target entity (e.g. object or aggregate), event type identifying what happened, event version for a time-based ordering of events, and the event payload itself.

In some cases, snapshotting (see section 2.10) is used to store the intermediate state. Snapshots can be stored with the corresponding last event processed. However, a more popular way is to store the snapshots separately. By not mixing other data structures with the events, each event can be kept as a small, simple, and self-contained piece of data. This enables persistence in many ways. See fig. 1.

The event payload is often stored as unstructured data [19][3], using an implicit schema interpreted by the application. This increases flexibility but reduces the availability of tooling for managing events [19]. However, as events often are stored as immutable data, the simple format enables easier distribution and caching of events.

Listing 1: Event data structure

```
{ :name "vessel-created"
```

```
    :data { :id 1
            :name "Vessel A"
            :built 2020 }}
```

2.5 Event streams

A stream contains all events for the entity or group of entities it is set to represent. E.g., for an aggregate. See fig. 1. This includes the creation, deletion, and manipulation of such entities. As it is a domain-oriented concept, it does not include technical events such as queries or event stream manipulation. Event streams can be implemented as one stream for the whole system or as multiple smaller streams (e.g., per aggregate). With multiple streams, implementation complexity increases, but flexibility increases as well (e.g., scaling and testing).

Events and event streams can have varying granularity. One event can either represent one or more application domain events. This can either be defined by the application designer or by the pruning process (see 2.11). An event stream with low granularity contains a limited number of compound events. An event stream with high granularity contains many small events. Both of these approaches can be used to generate the same application state. The difference is that the compound events are less able to be used to reconstruct intermediary states (states before current state). Also, the intent of the user becomes unclear as multiple actions are represented as one event.

2.6 State

In CQRS+ES, the application state (a complete or partial data model) can be reconstructed for any given point in the past or as current state. State hydration is the process of replaying events in event handlers so that current state can be reconstructed. E.g., multiple "period-added" and "period-removed" events will together comprise a complete vessel timeline. I.e., the event of events is run through a reducer, yielding a completely different data structure than its event inputs. Loading and re-processing all events every time a data structure is needed, can be costly in terms of resources and latency. To cope with this, state can be cached or events can be re-shaped to more effective data structures [25][9]. A common solution for caching, is to store pre-processed events in the form of rolling snapshots [26][25][20]. When hydrating state, the latest snapshot is located and newer events are applied on top of the snapshot. This yields current state. Fig. 2 shows this process where multiple events are transformed into current state.

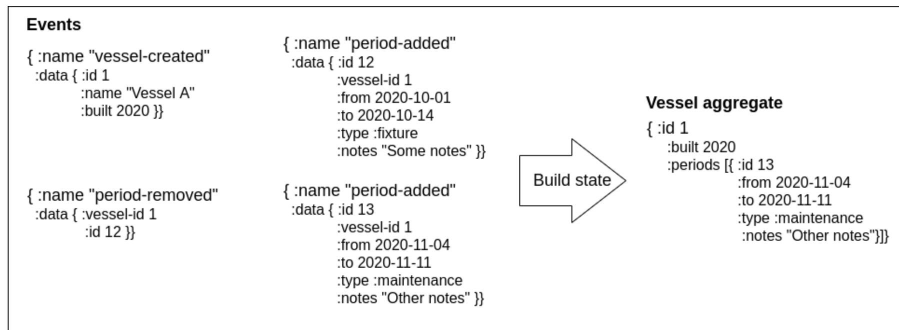


Figure 2: Building state from an event stream

2.7 Projections

Projections is a part of the query side of CQRS+ES. A projection can be any data structure derived from processing events. These data structures can be denormalized and thus optimized for their uses (e.g. for viewing user interfaces or reporting). Projections are managed through projection event handlers on the query side, receiving newly emitted events from the command side. As there is a delay between when an event is emitted and processed, projections are always eventual consistent.

2.8 Commands

Commands are initiated by the user, other systems, or the system itself. During execution, new events are emitted representing changes to the system. In event-sourced systems, these events are stored for further use. Thus, a command is only ever executed once. In command-sourced systems, each command is executed every time the system state is needed. To derive the same state every execution, each command must be deterministic. This requires storing parameters and temporal/mutable data with the command.

Querying across multiple aggregates inside a command might require loading all events in the system for a given aggregate type. A different approach is to query a projection and pass in relevant aggregate references in parameters to the command. This reduces load and enables commands to be deterministic. However, it must be taken into account that a projection is eventual consistent.

Listing 2: Command data structure

```
{ :name "aggregate.vessel/add-period"
  :parameters { :id 12
                :from 2020-10-01
                :to 2020-10-14
                :type :fixture
                :notes "Some notes" }}
```

2.9 Sourcing type

Sourcing type defines how the current state of an application is defined or derived. In CQRS+ES events are used to calculate current state. This is called event-sourcing. The events are then persisted and loaded when needed. A different approach is to use command sourcing. In this case, we only store the commands sent to the system and not the effect of them. By re-executing all commands, events can be produced again and used to hydrate current state. State is then incrementally constructed and held in memory. To enable this, the system must be constructed so that every command can be executed deterministically.

In CQRS systems it is not required to use event- or command sourcing. In this case, state can be stored and manipulated directly in its own model. This is outside of the scope of this paper.

2.10 Snapshots

Over time, a CQRS+ES system can accumulate a high number of events. Without any optimizations, performance might grind to a halt. A normal approach to this is to calculate a snapshot of the current state, and persist it [7][26][25][20]. As events newer than the snapshot appears, they are also applied to the snapshot.

2.11 Pruning algorithms

Event pruning is the process of removing redundant events, or re-shaping events to new data structures by combining or modifying them (e.g., removing or adding attributes). Current state for each aggregate will in many cases be the same before and after pruning. How much state can differ depends on application requirements. However, intermediary reconstructable states and preservation of intent will never be the same as when using the original events.

The pruning process can be viewed as a continuum where we have the commands (no events are kept) on one side and system state (snapshots) on the other side. Depending on system requirements, we can place our pruning algorithms somewhere in this range.

In listing 3, 4 and 5 we can see how the pruning process works. Listing 3 contains the original events of how a vessel was created, and started receiving position reports. Listing 4 shows the same vessel with redundant events removed, and combined. Listing 5 shows system state after combining events from either of the first listings. Thus, we have two sets of data with different characteristics that can be used to calculate the same end state.

Listing 3: Original events

```
{ :name "vessel-created"
  :parameters { :id 1
                :name "Vessel A"
                :built 2020 }}

{ :name "position-reported"
  :parameters { :id 1934
                :when 2020-11-09 11:27
                :type :position
                :lat 58.942
                :lng 5.253 }}

{ :name "position-reported"
  :parameters { :id 1934
                :when 2020-11-10 10:12
                :type :position
                :lat 59.431
                :lng 5.166}}

{ :name "position-reported"
  :parameters { :id 1934
                :when 2020-11-11 13:19
                :type :position
                :lat 60.41
                :lng 4.855 }}
```

Listing 4: Pruned events (from listing 3)

```
{ :name "vessel-created"
  :parameters { :id 1
                :name "Vessel A"
                :built 2020 }}

{ :name "position-reported"
  :parameters { :id 1934
                :when 2020-11-11 13:19
                :type :position
                :lat 60.41
                :lng 4.855 }}
```

Listing 5: Vessel aggregate after combining events from listing 3 or 4.

```
{ :id 1
  :name "Vessel A"
  :built 2020
  :last-position { :id 1934
                   :when 2020-11-11 13:19
                   :type :position
                   :lat 60.41
```

```
:lng 4.855 }}
```

In this paper, we will use seven different algorithms to build state. The first two (using original events and command sourcing) does not include any pruning step. The last five are described below. For these algorithms, events that are not needed can be deleted, and new events can be added to the event store.

Snapshotting algorithms construct current state from a set of events and persist it. When the aggregate needs to be loaded again, current state is retrieved directly. When new events arrive, the snapshot is updated.

Superseded algorithms remove previous events that have no effect on current state because a newer event overwrites its effect.

Bounded algorithm combines a partial snapshot with an event log of limited size. The partial snapshot is always updated by merging in new events so that the event log has a maximum number of events in it.

Probabilistic algorithms use an application-specific function to decide which events should be merged with earlier events. E.g geared towards the age of an event.

Hierarchical algorithms use an application-specific function to reduce granularity and reconstructability gradually as the number of events increases. E.g. by defining a set of consecutive windows of reconstructability.

Each of these algorithms can be divided into three categories: No pruning (unmodified event log and command sourcing), lossy pruning, and lossless pruning. No pruning builds state without any modification of the event streams. This enables us to handle the event log as immutable data by only adding new events, and never modifying or deleting existing data. Lossless pruning modifies the event streams so that the new current state is the same as when hydrating via the original events. The main difference is that we lose the ability to reconstruct intermediary states. Lossy pruning modifies the event streams so that the new current state is not the same as the original current state (i.e., with irrelevant parts removed).

2.12 Pruning execution

Applying pruning algorithms, as just described, requires a two-step process: Pruning initiation and event transformation. Pruning initiation defines how the pruning process is triggered (e.g., by application state, continuously or through user actions) and how it runs (e.g., runtime or externally). Event transformation defines how to operate the event store when modifying, removing, or adding events (e.g. upcasting, in-place transformation and copy and transform), and how event handlers support multiple versions of an event. E.g, via parallel stream mechanisms that preserve the original events, and keep pruned streams

separately.

2.13 Storage

We can divide storage into three areas: Event persistence, command persistence, and performance optimizations.

Events can be represented as simple self-contained data structures [26][25][20][4] containing metadata and event data. Metadata can consist of attributes such as event type and sorting data (timestamp or an autoincrement number). Events can be represented as unstructured data, leaving interpretation to the application. The goal is to capture enough information about the event to be able to calculate the effect it has on the application state.

Commands have the same characteristics as events: Self-contained structure with metadata (e.g. command type and timestamp), and command parameters as unstructured data. Command data structures need enough data so that they can be re-executed deterministically.

To improve performance, events, and commands can be augmented with preprocessed data such as aggregated data (e.g., summing up a value), relationships (e.g., entity ids), and cached replies from external services. To keep events and commands as self-contained data structures, we can put such performance optimizations inside the events and the commands.

To run a CQRS system we need both persistent and transient storage. Persistent storage is required for commands and events. For command-sourced systems, events are emitted and only kept in memory. For event-sourced systems events (and optionally commands) must be persisted. In both cases current state must be calculated and at least kept in memory.

2.14 Summary

In this chapter, we have detailed the building blocks and concepts of CQRS. We have also introduced the constraints CQRS and event-sourced systems operate within. By doing this we have also shown the relevance of our research questions for event-sourced architectures. In the next chapter, we will introduce related work in this area.

3 Related work

When building CQRS systems with a focus on performance, we need to look into three areas in the current literature. The event store handles the storage and retrieval of events. The state hydration process fetches events from the event store and combines them to build current state. The event log pruning process optimizes hydration by re-shaping data in the event store.

3.1 Event storage

The responsibility of the event store is to persist events and manage streams of events. This includes common operations on various levels of the store (store, stream, and event) such as add stream, add an event, and add attribute [19][22]. The DBMS system for the event stores can be any simple storage form according to [2]. E.g., a relational database, different nosql variants, directly on the file system, as flat files, or even in memory. Graph databases are also utilized as event stores [20][4]. The reason for the diversity comes from the simple structure of the event itself. A common representation is a simple schema-less compound data structure. This data structure usually contains data such as: Aggregate id, event id, event name, timestamp and domain-oriented event data [20][26][2].

As any persistent storage solution can be used for storing events we can find an overview of applicable storage solutions in [18]. This paper divides the field into two classes SQL and Nosql. For Nosql we also have subcategories such as key-value, column-oriented, document-stores, graph databases, and object-oriented database. In [25], the author tested coarse versus fine-grained events and found that compound events had better latency than fine-grained events. However, the paper was unconcluded as the difference might be caused by underlying performance problems. In [20], the author tested performance in SQL versus a graph database and concluded with better performance of the SQL database, caused by the time it takes to travel all relations versus a simple table scan. Adding clustering was also suggested as a method of improving performance. In [12], the author tested Nosql (MongoDB) versus SQL and found that the document store had better throughput than the relational database when issuing commands, and storing related events. However, it is a bit unclear how the events and commands were structured.

For modification of events, we need to look into both event upgrade techniques and deployment strategies. When events are pruned, they are essentially upgraded (transformed to a new version). This might be done in batch or continuously [7]. In [22][27][19], the authors suggest that events can be represented in a way that enables the event handlers to understand multiple versions of each event type. Alternatively, they can be upcasted or transformed. This transformation can be done when each event is needed or by batch transformation. Batch transformation can be handled by in-place transformation or by switching to a new event store. For deployment strategies, [22][27][19] suggest using either some variant of request routing to different versions of the system or to

use expand-contract mechanisms (supporting at first more attribute/variants, and gradually removing support for older versions).

3.2 State hydration

When state is built, a given set of events are processed in order. The organization of events can be done in multiple ways to improve performance and consistency. In [2][25][14][4], aggregates is used to co-locate events and state. This is done to reduce the number of events to read before reaching current state. I.e., by only reading a small set of entities. In [4], this concept is taken even further. As a part of improving write and read performance, each aggregate is placed on different shards, effectively partitioning the system data over many nodes to achieve better performance. A similar approach is suggested in [2] where bounded contexts are used as boundaries for partitioning data, effectively bundling together related aggregates and events.

For the actual hydration process, the method used or referenced in most papers is snapshotting [26][11][27][14][14][20][9][28]. Instead of processing every event a cache of pre-processed events are kept in some form, and updated as new events arrive (e.g., when aggregates are loaded or when the event log contains a predefined number of events). Snapshots can either stored as a global snapshot or per aggregate [2][8][6][3]. Instead of snapshots, current state of aggregates can be held in memory [9][2][5]. This variant requires all events to be loaded every time the system is started. If used in conjunction with aggregates, the number of events to be loaded can be reduced to the events relevant for the aggregate in question. A variant of the in-memory method is explored in [9], using command sourcing to build parallel realities. A simpler form of snapshots can be found in [3]. Instead of loading all events, aggregated values, and references can be placed inside event metadata, thus reducing the number of events to load. This is done simply by defining additional fields in the event data structures. However, this approach was not investigated further.

There is little literature available on performance testing of state hydration in CQRS+ES. In [4], the authors found that smaller aggregates were faster, and enabled higher concurrency as mutations were handled independently per aggregate. This enabled the use of small, isolated transactions. This also affected state hydration of aggregates, due to the reduced number of events to load. In the same paper, this was amplified by using sharding on the aggregate level. In [25], the author found that coarser-grained events were faster to process than finer-grained events. The trade-off is reduced granularity (fewer events). In [23], the author describes how Apache Kafka monitors the last read position of an append-only event log. This approach enables it to only read newly appended messages. This approach is not directly transferrable to CQRS+ES but the idea could be utilized in combination with in-memory snapshots.

3.3 Log pruning

Event log pruning is the process of rewriting existing events to new and more efficient data structures [7]. One of the core concepts in CQRS+ES is to store events as immutable pieces of data. This enables easy scaling (by log distribution), concurrency, and an append-only approach to persistence. Because of this, retroactive modification of events is not much explored. In [7] the authors acknowledge the importance of immutability but also advocates the opportunities available when not adhering to it such as maintainability and scalability (reduced number of events and storage requirements). Two solutions to keep immutability is presented in [17] and [27], where every modification either is managed as a new branch of the event log or where compensation events are added to the event log instead of modifying it. Similarly, branching with retroactive modification is presented in [6] as a way to execute advanced operations or simulations asynchronously.

A different approach to retroactive modification of immutable events is to avoid it as a whole. Command sourcing [17][9][7] can be applied without storing the events at all. It can't however, be expected to be the fastest method [7]. One command can produce more than one event [17]. When hydrating aggregates, every command ever given to the system (or aggregate) is executed again to hydrate current state. Retroactive modification of commands can be done through modification or deletion of commands, injection of new commands into the past, or by modifying command handlers to produce different events [15]. When applying command sourcing, every command must be deterministic. To avoid side effects in re-execution, some commands can be skipped or the command handlers can be modified. Introducing gateway interceptors can be used to catch calls to external systems [15]. Utilization of command sourcing can also be found in [17], where the author suggests simulation of different behaviors by re-implementing the command handlers.

Some of the pruning methodologies that can be applied instead of command sourcing are presented in [7]. All of them transform or remove events and can be applied to various scopes (aggregate, stream, or store), initiated by the domain model or the runtime platform continuously or reactively. The pruning can be based on various window strategies (events inside or outside a window defined by logical or wall clock time) or by bounded rings of n events. These approaches can be combined in a hierarchy with gradually reduced reconstructability. Distribution algorithms can also be applied to the pruning mechanism e.g. skewed towards age. Apache Kafka uses a topic compaction algorithm, similar to log-structured merge trees where the event log is scanned periodically, removing events whose state effect has been superseded by an event occurring later in the event stream [23]. Enabling the superseding pruning algorithm can also be done by applying reversal/compensation events [26][10], rendering previous events superfluous.

Removing unused event types is suggested as an approach to pruning the event logs in [2]. In [8], event log pruning is applied on a set of time-ordered events. The set is reduced to a snapshot and then persisted before all the events in

the set are deleted. However, care must be taken if any other applications rely on these events. Many papers on CQRS+ES suggest snapshots as a way to manage a high number of events (thus mostly in their theory/background section) [25][26][14][22] without adding the extra step of removing the original events.

Data conversion is an important part of log pruning. The authors of [27][19][2] discusses the pitfalls of version management for when events are upgraded (e.g., for adding new properties and removing properties). These challenges are relevant for our research question as there is a need to handle conversion between pruned and unpruned events as this is similar to event versioning. In [19] we are presented with two main upgrade methodologies: Batch and event-based. Batch means upgrading a part of the system at once using in-place transformation or conversion to a new store. Event-based utilize methods such as upcasting and support for multiple versions.

3.4 Summary

The event log pruning process reduces the number of events by transforming them into different events. A few large events seem to be faster than multiple small events for loading single aggregates. By exposing aggregate data and relations in properties smaller events might be able to provide enough data to proceed without loading all events (depending on the use case). Grouping of entities also seems to have a huge impact on performance. I.e., the organization of the event streams and which entities to hydrate. The performance of command sourcing is expected to be slow. A few papers suggest keeping current state in memory and reloading every event on startup. In a given case where there are not too many commands, it might be interesting to combine in-memory current state with command sourcing almost removing the need for slow communication with a persistence mechanism. How the persistence layer can affect performance in CQRS has been tested in a few papers but it does not seem to be conclusive. However, there seems to be a consensus on how events should be stored (as simple entities with some metadata and unstructured event data).

A different issue is the complexity of adding log pruning to CQRS+ES. To enable log pruning the principle of immutable data must be broken and we need to support different upgrade strategies for the event store and the CQRS+ES software. In addition, the software needs to understand different representations of the same current state. One where the event log is not pruned and one where it is pruned.

4 Methodology

We want to understand the effect of pruning in CQRS+ES systems. Because of this, we can't test pruning algorithms in isolation. We then need a complete working CQRS+ES system instead. Such a system can be adapted and tested under various configurations. After establishing a base test case without pruning it is possible to introduce various pruning algorithms and measure system performance on various data sets. This requires full access (code, data, and persistence) to a CQRS+ES system as cross-cutting changes must be implemented within a limited time-span. As there are few such systems available for us with this degree of freedom, a new CQRS+ES system within a fitting business domain was implemented from scratch to run tests on. This chapter describes this process in detail.

4.1 Simulation case

The simulation case is a subset of a business application managing the PSV (Platform Supply Vessels) fleet in the North Sea for vessel brokers. The end goal is to enable instant bidding on short term contracts (spot fixtures) to deliver liquids and goods to and from offshore installations. To be able to do this the system needs to be aware of vessel availability. Vessel availability is defined by the vessel's current and future activities (on maintenance and on contract) and its location relative to the operational area of potential contracts. A vessel too far away might not be able to meet a tight deadline. This problem can be resolved by recording a timeline of vessel activities and positions.

The system data model has two main entities: Vessels and fixtures. The vessel entity contains vessel data such as reported locations (received via the AIS positioning system), represented as a list of positions, and a timeline represented as a list of periods. See listing 6. The fixture contains contract data (from date, to date, and assigned vessel id), and must be assigned to the vessel that won the contract. Other details have been omitted (such as vessel specifications, and other timeline period types) to be able to focus on the pruning process.

To manipulate the domain model, a set of commands can be sent to the system. E.g. "create-vessel", "create-fixture", "report-position" and "add-period".

Listing 6: System entities.

```
Fixture entity:

{ :id 1
  :from 2020-11-25 12:00
  :to 2020-11-27 18:00
  :vessel-id 1 }

Vessel entity:
```

```
{ :id 1
  :name "Vessel A"
  :built 2020
  :last-updated 2020-11-25 14:36
  :periods [{ :id 2
              :from 2020-11-25 12:00
              :to 2020-11-27 18:00
              :type fixture }
            { :id 1
              :from 2020-11-02 23:00
              :to 2020-11-01 06:00
              :type fixture }
            ...]
  :positions [{ :id 1935
                :when 2020-11-11 13:19
                :type :position
                :lat 60.41
                :lng 4.8 }
              { :id 1934
                :when 2020-11-19 14:29
                :type :position
                :lat 59.41
                :lng 4.855 }
            ...]
 }
```

4.1.1 Model parameters and data generation

Population parameters for vessels were pulled from [24] p. 19 as it contains an overview over the PSV market in the North Sea from 2007 to 2016. Also, AIS reports for vessels in Danish waters were obtained from [1], giving details on expected report types and frequencies. Based on these parameters, a randomized model generator was set up to generate smaller and larger data structures as candidates for pruning and persistence tests.

To generate randomized data within the bounds of the population parameters, a set of test parameters were defined as a basis for the test data generator. As command generation and processing take quite some time on large data sets, the whole range of values was not tested. As an example, our simulations initially took between 6-9 hours to execute (all tests on 22 vessels over 6 years). Following are the default parameters (population parameters and comments in parentheses):

- Num vessels: 12 - 31 (10% of the population (120 - 313 vessels))
- Vessel age: 0 - 11 years (Population = 0 - 37 with an average of 6-9 years)

- Preferred vessels: 40% (Arbitrary percentage of the vessels are preferred and thus each given an increased weight of 1.3 for being selected)
- Vessel utilization per month: 31% - 72% (How much of the month the fleet is on contract)
- Contract length: 3 - 30 days (Length of each assigned contract. Combined with utilization above. Arbitrary values within the range of spot market contracts at max 30 days)
- AIS event resolution: 24 hrs (One AIS report is added to each vessel every 24 hours. Actual resolution can excess 18.000 reports per day but such a resolution is not required in this case)
- Simulation interval: 2021-01-01 - 2031-12-31 (The period the simulation runs)

By running these parameters through the data generator, a set of fixtures were generated based on the simulation period, the number of available vessels (based on age), and utilization. Then each vessel was generated with no positions and with an empty timeline. Fixtures were then assigned randomly to vessels according to age and popularity. In the end, AIS reports were generated from the build year of each vessel and, to the end of the simulation (independently of any fixtures on the vessel).

To understand the effect of the parameters on the data model, a sensitivity analysis was also applied on aggregate loading time vs. utilization and AIS reporting frequency.

4.1.2 Model interaction

The vessel fleet system is expected to have many more reads than writes. One write (or command execution) only operates on one aggregate at a time, and each vessel might receive 1-2 commands per day. This results in 1-2 aggregate loadings per vessel per day as reads are separated to denormalized views. Even if the aggregate loads are very few, they must be within reasonable response times (with or without pruning). We can use our simulation model to test this capability by varying the frequency of AIS reporting, utilization percentage, and vessel age. By doing this we gradually increase the load on the system.

To be able to interact with the whole data model, two read models were defined as projections. One as a list of vessel ids, and one as a list of fixtures ids. By querying these projections, the test execution software and the pruning software can interact with all aggregates in the system by loading them one by one.

4.2 System design

The CQRS+ES system was implemented in Clojure using maps, lists, and vectors to represent events, commands, and aggregates. Clojure is compiled to Java and runs on JVM. To persist data, we implemented support for Mysql, MongoDB, and EDN flat files through a common interface using polymorphism. This chapter explains the implementation details of this system, starting with the architectural overview. Instructions for how to install the system can be found in appendix A.

4.2.1 Architectual overview

Figure 3 shows a high-level architectural view of the CQRS+ES system and its components for pruning and running tests.

The workflow is as follows: **(1)** The test runner picks up default test parameters and randomizes number of vessels, vessel age and utilization (according to population parameters). After this, a set of commands are created to build up the complete simulation data model. E.g., "create-vessel" for each vessel, "report-position" on every AIS interval, and "craete-fixture" for each fixture. **(2)** The test runner calls the command dispatcher sending in a command data structure. **(3)** The command dispatcher finds the correct command handler, and executes the command. **(4)** The command handler then loads relevant events from the event store, and hydrates the aggregate by applying events to it. **(5)** The command is then executed on the aggregate, producing new events, and **(6)** persisted by the command handler in the event store. (7) Projection event handlers are notified so that they can update projections. (8) If applicable (e.g., for the "vessel-created" event), the projection is updated. **(9)** With all events in place in the event store the pruning process is triggered (if needed). **(10)** The pruning process manipulates the event store according to each pruning algorithm. (11) All vessel ids are then retrieved by querying the "all-vessels" projection. Each aggregate is then loaded by triggering a dummy command that only loads the aggregate. **(12)** Measurement data is then added to a file used as an input for the analytics executable. See **??** and figure 4.

To test multiple event stores, the same set of commands (generated in step 1) are executed on each event store.

4.2.2 Aggregates

There are two aggregate root types in the system. One to represent fixtures and one to represent vessels. See listing 6. Each aggregate is represented as a Clojure map with properties defined as scalar values and vectors. As Clojure is a functional language, the aggregates are only data. Code that handles the aggregates is placed in their own namespaces. This code is responsible for receiving commands and returning new events. It is also responsible for taking

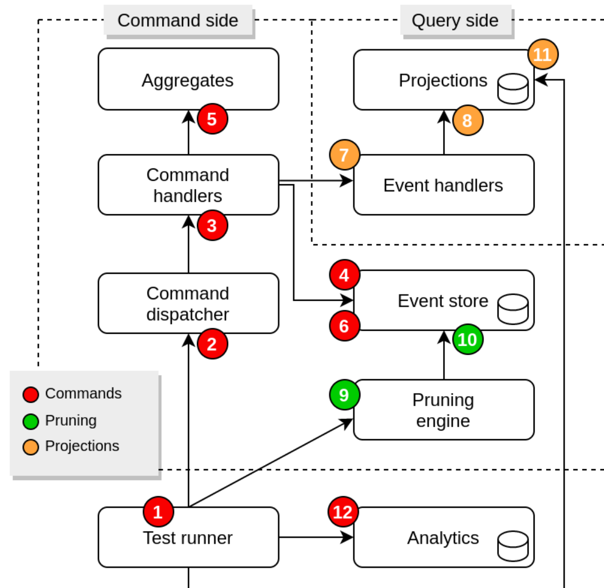


Figure 3: System partitioning and workflow

the current state of an aggregate and a list of new events returning a new version of the aggregate with the events applied. Relations between the aggregates are represented as UUIDs and used in all interactions with the aggregates (e.g., for lookup and in command handlers).

4.2.3 Commands and events

A command is initially passed to the command dispatcher as a set of parameters and thereafter persisted as a Clojure map (see 4.2.5). Listing 7, shows an example map for a command. The map contains enough data to be able to use it as a basis for command sourcing (re-execution of the command). To execute a command, a fully qualified function name is required to identify the namespace and the function responsible for executing the command. In addition, an aggregate-id and an aggregate type are required for loading state. At last, a map of all command parameters must be provided. The command parameters can be any data structure but for the vessel system, they are of similar size (only a few scalar values).

The event data structure is represented as a Clojure map containing the name of the event and the data accompanying the event. Listing 1 shows a complete event map. The event contains all data needed to pass it to the state hydration process to recreate current state. As for commands, event parameters are of similar size throughout the system (only a few scalar values).

Commands and events are all implemented using UUIDs to identify aggregates

and relations between them. These UUIDs are generated outside of the command handling process to enable deterministic execution of commands. See more on this in 4.2.8.

It should be noted that we used a shorthand notation of :n and :p for :name and :parameters in the code to reduce storage space. Also, in some examples, UUID is written as a single number to enhance readability.

Listing 7: Command data structure

```
{ :name "aggregate.vessel/add-period"
  :parameters { :id 12
                :from 2020-10-01
                :to 2020-10-14
                :type :fixture
                :notes "Some notes" }}
```

4.2.4 Command dispatcher

The responsibility of the command handlers is to execute commands. There are three dispatch functions for executing commands, all requiring an aggregate id (or empty on aggregate creation), aggregate type, a fully qualified function name for the command handler function, and command parameters as a map. The "dispatch-es!" function executes a command using event sourcing to build state. The "dispatch-cs!" function executes a command using command sourcing to build state. The "dispatch-aggregate-cache!" function uses an in-memory cache of aggregates to avoid building state multiple times in a row. This cache is only used for executing commands to build the initial data model before testing starts.

Before a command is executed, state is built (see 4.2.5). Directives on which event store to used are passed as a configuration map to the dispatch function. Then the command function is located and called. The location mechanism converts the fully qualified function name parameter to a symbol before resolving it. When calling the function, it is passed the old aggregate and the command parameters. See listing 8. New events (zero or more) are then passed back to the command handler and persisted in the event store. At last, the events are emitted to the projection event handlers so that they can update their projections.

Listing 8: A command handler for the vessel aggregate

```
(defn report-position [aggregate params]
  (vector
     {:s (:id aggregate)
      :n :position-reported
      :p params}))
```

4.2.5 Streams and state hydration

The command dispatcher passes the aggregate id and the aggregate type to one of three state-building functions: "load-aggregate-es-", "load-aggregate-cs-" or "load-aggregate-cache-". These functions use either an event store or in-memory cached aggregates to return state. In the case of an event store, a call is passed to the event store, executing a query for either a command or event stream. As both commands and events are persisted as streams, there are two streams for each aggregate. The event store then returns a sequence of all events or commands from when the targeted aggregate was created. Dividing state data into small co-located streams reduces the amount of data to load and process per state reconstruction.

Command sourcing uses a similar algorithm as explained in 4.2.4. The main difference is that nothing is persisted. Building state is done by iterating over the set of returned commands: Starting with an empty aggregate, the first command is executed. The returned events from the command are then applied to the empty aggregate, returning an updated aggregate. The process is repeated passing in the updated aggregate instead of the empty one. After executing the last command, the aggregate is returned with current state.

Event sourcing follows a similar approach as command sourcing but instead of having to generate the events by re-executing the commands, they are read directly from the event streams. The process is as follows: Starting with an empty aggregate, apply the first event on the aggregate. Repeat the process for all events, passing in the gradually updated aggregate for each iteration.

As shown, events are always used to build state. Either directly (when using event sourcing) or indirectly (generated by commands when using command sourcing). Listing 9 shows the implementation of the state hydration function for the "vessel-created" event.

Listing 9: A state building function for the vessel aggregate

```
(defmethod apply-vessel-event :vessel-created
[aggregate {{:keys [id name built]}
             :p :as event}]
  (assoc aggregate
     :id id
     :name name
     :built built
     :periods []
     :positions []
     :last-position nil
     :last-updated nil))
```

4.2.6 Event store

An event store needs to operate on the store, stream, and event level. However, as this event store was implemented from scratch, we only added support for the bare minimum of operations to reduce development time. E.g., we have support for deleting events, but we don't have support for manipulating event properties (e.g., merging events and removing properties). This is instead achieved by combining other operations (e.g. "delete-event!" and "persist-event!"). We also added functions for backing up and restoring an event store to optimize test preparations.

The event store is implemented as a set of multimethods. A multimethod is a runtime polymorphic construct that uses a dispatch function to determine which function to execute. The dispatch function receives the function parameters and compares the returned value to dispatch values of defmulti functions with the same name. See listing 10 which shows how the ":type" key is read from the connection map to persist an event to Mysql. In addition to this function, each event store support other operations: "delete-streams!", "delete-stream!", "stream-" (load a complete stream), "delete-event!" and "delete-events!". There are also polymorphic functions for supporting multiple projection storages via "projection-insert!" and "projection-query-". By combining these functions, we can build the test model, perform pruning and execute tests.

The vessel system supports three persistence mechanisms: Mysql, MongoDB, and serialized EDN data structures (referred to as EDN flat files or EDN files from now on). Each of these mechanisms has its own implementation of each of the multimethods just mentioned, but with different dispatch values (":mongodb", ":mysql" and ":multi-file-edn"). There is only one implementation for the projection store (Mysql) as it is not important for running the tests.

Listing 10: Polymorphic persistence support

```
(defmulti persist-event!
(fn [connection event e?] (:type connection)))

(defmethod s/persist-event! :mysql [connection event e?]
  (clojure.java.jdbc/insert!
     (s/connection-map connection)
     (table-name e?)
     (s/event-map connection event e?)))
```

4.2.7 Persistence

Any persistence mechanism can be used as long as the functions in 4.2.6 can be implemented on top of it. A classification of database management systems can found in [18]. Ideally, test coverage should span all of these database classes for completeness. To reduce development time we only implemented a subset of them: Relational (Mysql), document (MongoDB), and flat file (serialized

EDN vectors). To be able to compare their implementations, a few design guidelines were defined. A complete stream must retrievable via a single query. Commands must be handled as events. Events and commands must be separate streams. Other persistence design decisions must favor read performance of a single stream. Snapshots and other aggregated events must be stored as events.

Mysql is a relational database. In our implementation, we defined two tables, one for events and one for commands. See listing 11. Each tuple consists of a timestamp (t), stream/aggregate id (s), event or command name (n), and parameters (p). On persistence, the parameters column is serialized to text. The process is reversed on retrieval. To enable this the EDN (Extensible Data Notation) format is used. EDN is a subset of the Clojure data types and is directly usable in Clojure. As Mysql has a max row size of 65535 bytes the maximum size of the parameter map is set to 21000 bytes taking into account other columns, and UTF-8 storage requirements (more than one byte for some characters). To support larger parameter maps (e.g. for snapshots) the pb column is used to store the map as a blob outside the table page files. For fast retrieval and sorting, we have two indices. The first index on stream and event type enables fast retrieval of a complete stream. The second index on timestamp, stream, and event type makes sure sorting is already available when fetching a complete stream. Projections are only stored in Mysql. See listing 12 for how they are defined. For connectivity, we used the clojure.java.jdbc library.

Listing 11: Mysql event and command tables.

```
create table event (
 t bigint not null,
 s char(36) not null,
 n varchar(50) not null,
 e int not null,
 p varchar(21000) default null,
 pb mediumtext default null
) ENGINE=MyISAM;

create index tse on event (t,s,e);
create index se on event (s,e);
create table command like event;

Properties:
t = timestamp
s = stream (UUID)
n = name (event or command)
e = is event?
p = serialized parameter map
pb = serialized parameter map as blob
```

Listing 12: Projections stored in Mysql.

```
create table all_vessels (
  aggregate char(36) not null
) engine=MyISAM;
```

```
create table all_fixtures (
  aggregate char(36) not null
) engine=MyISAM;
```

MongoDB is a document database. Each document is a self-contained arbitrarily nested data structure. On the top level, we have collections of documents. Each collection represents a set of similar documents. The hierarchical nature of documents enables simple persistence and retrieval of a complete record. Our implementation utilizes this capability by embedding a whole stream in each document. See listing 13. The "_id" property is the aggregate id in binary format and identifies a stream. It also doubles as a default unique index for the collection, enabling quick retrieval of the document. A document property "entities" contains all events (or commands for the command collection), ordered by time. MongoDB stores documents as JSON in a binary format called BSON. To operate the MongoDB database we used the Monger library that wraps the MongoDB Java driver. This library also converts the BSON documents to and from Clojure data types.

Listing 13: MongoDB event/command collections and example event stream

```
db.createCollection('event');
db.createCollection('command');

{ "_id" : BinData(3,"ukZdSJrK9iSmxFCJC/omug=="),
  "s" : "24f6ca9a-485d-46ba-ba26-fa0b8950c4a6",
   "entities" :
   [ { "t" : NumberLong("1606206429413"),
       "n" : "vessel-created",
       "p" :
         { "id" : "24f6ca9a-485d-46ba-ba26-fa0b8950c4a6",
           "name" : "Some vessel 0",
           "built" : NumberLong("1843336800000") } },
         ...]}

Properties:
_id = stream (binary UUID)
s = stream (human readable UUID)
entities = vector of all events or commands for a stream
t = timestamp
n = name (event or command)
p = parameter map
```

Multi file EDN is a flat-file persistence mechanism designed for this project. Flat file storage is a broad term for data storage solutions that are oriented around manipulating files containing data records stored as text, or in binary form. The variant implemented in this project uses serialization to and deserialization from text, using the EDN data format. Each stream is persisted as two separate files. One file for events and one for commands. Both files contain

the aggregate id in the file name for fast retrieval. On persistence, each event or command is appended to the end of each file. See listing 14. Before deserialization, vector ([]) characters are prepended and appended to the file content. This enables EDN to parse the text as a vector of events or commands ordered by time, without any other manipulation.

Listing 14: Flat file content and file names

```
File content:
[...
{ :t 1606206574081,
  :n "vessel-created",
  :p "{ :id \"24f6ca9a-485d-46ba-ba26-fa0b8950c4a6\",
       :name \"Some vessel 0\",
       :built 1843336800000}" }
...]

File names:
24f6ca9a-485d-46ba-ba26-fa0b8950c4a6-e
24f6ca9a-485d-46ba-ba26-fa0b8950c4a6-c

Properties:
t = timestamp
n = name (event or command)
p = parameter map
```

4.2.8 Sets and projections

CQRS loads and operates on one aggregate at a time. Even when working with a process (saga) one and one aggregates is loaded in sequence. Because of eventual consistency, and deterministic execution (for command sourcing), identifiers (UUID in this system) must be passed in as a part of the commands and not queried after inside the command handlers.

In this system, we need to operate on all aggregates in sequence to be able to execute the pruning process. This was resolved by implementing two projections stored in Mysql only. One projection contains a list of all vessel UUIDs, and another one contains a list of all fixture UUIDs.

4.2.9 Pruning process

Each pruning algorithm has a single function to initiate it from the testing process. These functions require at least two parameters: A connection map (e.g. for Mysql) and a list of vessel UUIDs from the "all-vessels" projection. Some of the algorithms are also passed parameters that define pruning boundaries. E.g., "prune-bounded!" needs to know how many events to keep before creating a snapshot.

The general pruning process is to iterate through the list of vessel UUIDs, loading either their complete stream and/or current state. By inspecting the returned data the algorithms decide which events to remove, or add. E.g., for snapshots, all previous events are removed, and a "snapshot-created" event is added instead.

When the pruning process has been completed, aggregates can be loaded via the ordinary "stream-" functions. When it is time for testing the next pruning algorithm, the original event store is reconstructed from a copy.

In the case of command sourcing and pure event sourcing (using original events), no pruning process is initiated.

4.2.10 Pruning algorithms

The pruning algorithms are responsible for reducing the number of events in the event store. Depending on the use case, events can be removed, added, or combined to yield a current state with different granularity and degree of reconstructability. In total, we implemented five pruning algorithms. See chapter 2.11.

Below we describe the implementation of each algorithm:

Snapshotting Our implementation builds current state from a complete event stream and persists the result as a new event type called "vessel-created-full-snapshot". The complete current state can then be found in the parameter map of the event. All other events are then deleted from the event store. When the system needs to load state it receives only the snapshot event. On state hydration, the whole parameter map is returned directly as the aggregate.

Superseded The system has two events types that can be pruned this way. The "vessel-updated" event is emitted when a vessel is created or a period is added to it. In the vessel aggregate, the newest value is kept in the ":last-updated" key. Previous values are overwritten. The second event type is the "position-reported" event. The newest AIS position report is placed in the vessel map on the ":last-position" key. The previous value is overwritten. The implementation of this algorithm removes all "vessel-updated" and "position-reported" events, except the last one of each.

Bounded Our implementation uses a variant of the snapshot pruning algorithm to build the initial snapshot. The difference is that the snapshot is built by including all events, except the n last ones. In our case, we create a snapshot and keep the last 100 events in the event store. This means that we never load more than 101 events for the vessel aggregate to hydrate current state.

Probabilistic Our implementation of this algorithm does not merge events but remove events that can be considered irrelevant. All "vessel-updated" and "position-reported" events are removed except from the last one of each. In

addition, only ongoing or future vessel periods are kept. This means that older "period-added" events are removed from the vessel streams. The focus is on future vessel availability.

Hierarchical Our implementation uses a domain aware function to reduce the number of AIS reports to keep in the event store. This function keeps all reports newer than 30 days. For events between 30 and 59 (including) days, every second is dropped. For AIS reports older than this, we only keep every fourth. This enables the application to retain full details on recent movements, while still keeping some details on older positions.

4.3 Simulation process

The simulation process is the coordinated execution of all the previously described system parts in a way that enables us to measure, aggregate, and report results. This section contains a description of how it is implemented and executed.

4.3.1 Process

The vessel system runs on a single thread executing every step in sequence. Test execution starts with passing in a map with parameters. This map contains event store connection maps and parameters for data model generation and pruning. Some of the parameters for the data model generation (number of vessels, age of each vessel, vessel utilization, and contract length) are generated by using the Clojure rand and rand-int functions within a predefined range. Appendix B contains this parameter map. Population parameters were extracted from [24] page 19 and used to define vessel parameters: Age (on average 6 years), utilization (31% - 72%), and quantity (12 - 31, 10% of reported population). The simulation runs over 11 years (same as vessel age). The reason for not including more vessels and a longer simulation period is to reduce the time it takes to generate and execute commands to build the data model. The main focus is on loading current state for a single aggregate.

The simulation process is as follows (see fig. 4): **(1)** Remove all data in the projection tables. **(2)** Generate a set of command data structures that will produce the intended data model when executed. E.g., create-vessel 1, create vessel 2, report position to vessel 1, and so on. This includes assigning fixtures to vessels according to age and popularity. Commands are generated within the range defined as start and end dates in the parameter map. End date also represents current time. **(3)** Each event store is then tested in this order: MongoDB, Mysql and EDN file storage. The following steps describe testing a single event store. **(4)** Data in current event store is deleted. **(5)** Every command from step 2 is executed through the command dispatcher. Execution is done through a special command dispatcher that takes an aggregate as a parameter instead of loading state via the event store. These aggregates are

kept in memory and gradually updated as testing progress by applying new events to them. New events and commands are persisted to the event store. **(6)** The resulting events and commands are copied to be able to roll them back between each test. **(7)** Testing starts with the pure event-sourced variant (no pruning), continues with the five pruning algorithms, and ends with command sourcing. The following steps describe the test step. **(8)** Roll back copied events. **(9)** Stop execution for 2 minutes to let the OS, DBMS and GC to finish eventual longer running tasks. **(10)** Execute the pruning process (or skip for no pruning and command sourcing). **(11)** Cooldown 2 minutes. **(12)** Load vessel UUIDs from the "all-vessels" projection. **(13)** Execute and measure the "load-aggregate-es-" or "load-aggregate-cs-" functions 1000 times for each vessel aggregate in sequence. **(14)** Return load measurements to be compiled into a map also containing the number of events, event sizes, and other metadata.

To make sure the results are drawn from data models representing a varied set of conditions the simulation is repeated five times. Each time a new data model is generated and the same set of tests are executed on it. Results are then reported on all vessels generated in all simulations.

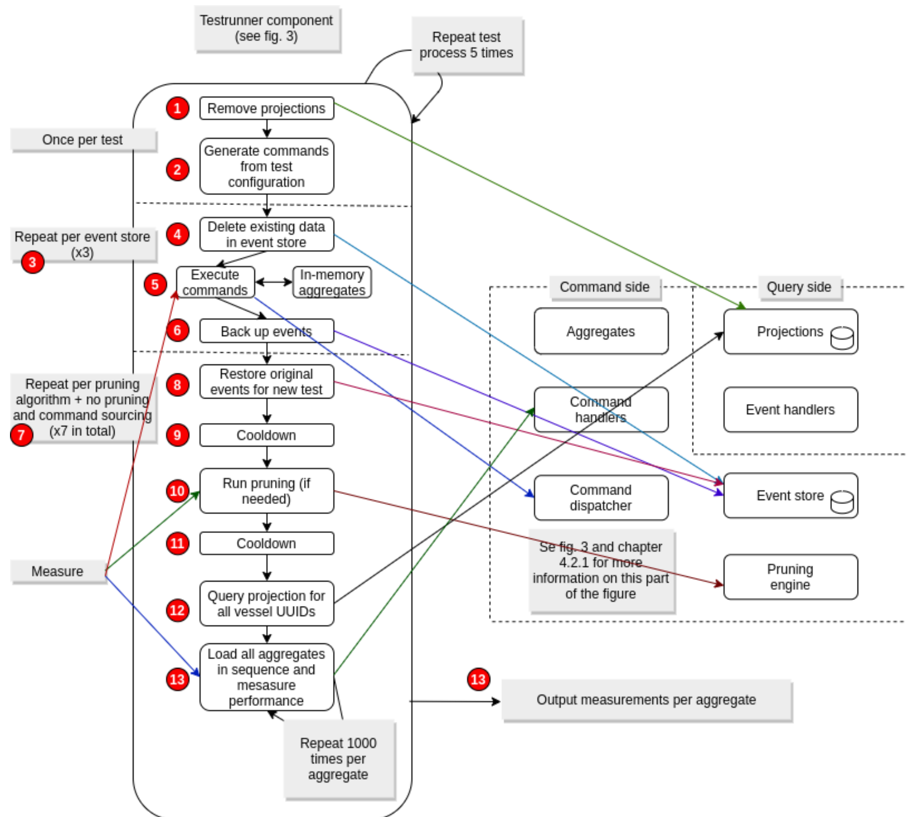


Figure 4: Testing process

4.3.2 Parameter sensitivity

To help identify the characteristics of the output simulation model we did a preliminary sensitivity analysis to see which model input parameters affect the output simulation model and the simulation. It is expected that the number of events and their size will be the main factors defining load time to reach current state. As only one aggregate is loaded per command the focus is on a single aggregate.

In the vessel aggregate, two entities can grow rapidly in quantity; position and period. These are defined by the "position-reported" and "period-added" events. The size of each event type is similar. The "period-added" event occurs less frequently as it is defined by one or more calendar days per event. The "position-reported" event occurs every day. AIS reports are sent from a vessel up to thousands of times every day. Since we only need to be aware of candidate vessels in a certain area, the rest is dropped. By varying the AIS report interval input parameter, and the utilization percentage we can measure how it affects loading performance.

Running these simulations follows the same steps as in 4.3.1 but only for one vessel, and with no pruning. To be sure the effect of variation can be attributed to one single parameter there are two parameter sensitivity tests. One for the utilization parameter and one for the AIS report frequency parameter. In addition, vessel age is fixed to 6 years for both tests, and utilization is fixed to 50% for the AIS test. All other parameters are the same as in 4.3.1.

4.3.3 Measurements

When testing the performance of each pruning process (including pure event sourcing and command sourcing) it is the state hydration process that is measured, not the command handling process. This process starts with fetching a complete event or command stream from the current event store. Thereafter the events are fed in sequence through an apply function gradually building state. It is only the loading of vessel aggregates that are tested.

Measurements are done using Java.lang.System.nanoTime(). Tests done in [21] shows a measurement granularity of about 30 ns with a call time of time of 15 - 30 ns on low-end hardware. To measure expected memory consumption we use the MemoryMeter Java library to calculate the size of aggregates, events, and commands.

4.3.4 Software stack and hardware

To run simulations we used a dedicated Linux server running Ubuntu 20.04 with 12 GB of RAM, NVMe SSD persistence, and a Xeon 3.07 GHz processor (4 cores). The simulation software written in Clojure was compiled to Java

bytecode and executed on the OpenJDK runtime environment version 1.8.0_275. Mysql server version 8.0.22-0ubuntu0.20.04.2 and MongoDB version 4.4.1 were installed alongside on the same server, both community versions.

4.4 Summary

The vessel simulation case enables testing of a realistic and randomized workload for the CQRS implementation. Instrumentation of a command generator creates commands that produce events with a target data model in mind. These commands are executed through the normal workflow of the CQRS system. By developing a generic persistence interface we can investigate the performance characteristics of different persistence mechanisms. The same interface is also used to prune the event stores. State loading algorithms in combination with pruning algorithms enable us to test state loading with different characteristics and design implications. By implementing a CQRS+ES system from scratch we can control the testing process in detail. The state-building process can then be executed without interference from the command handlers and the pruning process. This enables us to measure what we intend to measure. The next chapter presents the results from running tests on this system.

5 Results

This chapter presents the results found by following the methodology in the previous chapter. The process from 4.3.1 (see fig. 4) was repeated five times and results were thereafter aggregated. A summary of this process is a follows: Generate data model, execute pruning, test loading, and report measurements and metadata. The total count of all generated vessel aggregates in all tests was 95, an average of 19 per test execution. See table 2. Reported measurements have been rounded down. For all tables in this chapter, the "No pruning" algorithm denotes pure event sourcing without any pruning applied (i.e., the original events that the other algorithms reduce by pruning). This is the base case to compare other methods to. It must be noted that all testing revolves around vessels and not fixtures.

5.1 Command execution

The system simulation starts with the generation of commands to populate the simulation data model. Command generation is guided by the test parameters (see appendix B). Every one of these commands is then executed, and the result (events and commands) persisted in each event store.

The measurements only include command execution, excluding command generation and state loading. See fig. 4, step 2 and 5 and the explanation of the steps in 4.3.1. Command execution normally includes the time it takes to load the target aggregate and persist the resulting events and the command itself. In this simulation, we used an in-memory cache of all aggregates. Instead of loading events from the event store, we gradually updated the in-memory version. The measurements then comprise the time it takes to look up the aggregate in memory and persist one or more events and one command. It should also be noted that no command produces more than two events.

5.1.1 Data

Table 2 presents measurements from the command execution process. The "No. vessels" contains the number of vessel aggregates to generate. The "Total time (s)" column contains the time it took to execute all commands to generate the test data model. The "Per command (ms)" column contains the mean command execution time for a single command on average. The column "Commands / s" contains data on many commands the system was able to execute per second on average. The "No. commands" column contains data about how many commands were executed in total.

	No. vessels	Total time (s)	Per command (ms)	Commands / s	No. commands
MongoDB	95	2,005	45.33	22.06	44,229
Mysql	95	1,738	39.29	25.45	44,229
EDN files	95	56	1.27	789.80	44,229

Table 2: Mean command execution time

5.1.2 Results

Executing 44,229 commands enabled us to achieve a throughput of 22.06 commands / s for the MongoDB event store, 25.45 / s for Mysql, and 789.80 for EDN files. This means that the EDN file store outperformed the MongoDB event store with a factor of 35.8 and Mysql with a factor of 31.03. Comparing MongoDB and Mysql we can see that Mysql outperformed MongoDB with a factor of 1.15, enabling Mysql to execute 3.42 additional commands per second. For all three event stores, a single command took between 1.27 ms and 45.33 mm on average to process with in-memory aggregates.

5.2 Event log pruning

The pruning process removes and combines events according to the algorithms presented in 4.2.10. The process executes before testing starts and processes all aggregates in sequence. This step is skipped for the no pruning tests and command sourcing. The pruning algorithms follow this general form: Load the event stream. Inspect the stream according to the algorithm. Remove unwanted events. Add new events (e.g. a snapshot event). See also fig. 4, step 10 and 4.3.1.

5.2.1 Data

Table 3 presents time spent executing the event pruning process. The leftmost column is the name of the applied pruning algorithm. The following three columns contains mean pruning time (ms) per aggregate. Empty values indicate that pruning is not needed.

Table 4 contains data about the content of the event stores before and after the pruning algorithms have finished. The base case is the "No pruning" row which tells us the state of the event store without any pruning applied. Every following pruning algorithm uses this data set to perform pruning (see step 8 in fig. 4). The following rows contain data gathered from the event stores after each pruning process has been completed.

Table 4 has the following rows: "Mean no. events per stream" contans the

Pruning algorithm	MongoDB	Mysql	EDN files
No pruning	-	-	-
Superseded	48,593.26	42,985.43	783.94
Bounded	2,204.72	2,178.13	133.69
Probabilistic	48,549.38	44,356.35	806.90
Hierarchical	19,437.25	10,540.10	890.22
Snapshot	105,952.45	44,693.39	1,098.18
Command sourcing	-	-	-

Table 3: Mean aggregate pruning time (ms)

mean number of events in each stream. "Mean event size (KiB)" contain the mean size of each event (stream total size divided by the number of events). "Mean stream size (KiB)" contains the mean total stream size (all events in the stream). Building an aggregate requires one complete stream. As each event store has some variations in size, the two last columns are an average over the three event stores. It must also be noted that the size measurements are from the event data structures in memory and not in storage.

5.2.2 Results

The numbers in table 3 show us that the pruning process time ranged from 783.94 ms to 105,952.45 ms per aggregate across all event stores. EDN files were the fastest event store with a range between 133.69 ms and 1,098.18 ms, a factor of 11.84 to 96.48 faster than MongoDB and Mysql. Mysql outperformed MongoDB on average by a factor of 1.25. The bounded algorithm outperformed the other algorithms by factors from 5.86 to 48.05 across all event stores. If we exclude the bounded algorithm the performance was more similar, factors from 1.03 to 5.45.

As the pruning algorithms are executed the characteristics of the streams change as well. Table 4 shows that the no pruning variant with 2,328.84 events was reduced on average by 83.14%. The memory requirements were reduced by 84.97% on average. If we exclude the algorithms that produce a different current state than originally (probabilistic and hierarchical) the number of events was reduced by 97.48% on average. Memory requirements were reduced by 83.53% on average.

Pruning algorithm	Mean no. events per stream	Mean event size (KiB)	Mean stream size (KiB)
No pruning	2,328.84	0.34	779.62
Superseded	75.32	0.43	32.10
Bounded	99.97	5.64	564.11
Probabilistic	3.47	1.08	3.74
Hierarchical	1,782.97	0.33	595.86
Snapshot	1.00	554.60	554.60
Command sourcing *	2,316.64	0.34	794.17

Table 4: Events after pruning * From a limited set of tests (see 5.3.1)

5.3 State hydration

Measuring the state hydration process includes loading all events for each vessel aggregate in the event store, and passing them through the hydration functions. For command sourcing, commands are loaded from the event store instead. After executing each command, state can be gradually built by applying the returned events from each command.

This process was repeated 1000 times and used to calculate the mean load time in ms for a single vessel aggregate. This process is depicted in step 13 in fig. 4.

5.3.1 Data

Mean aggregate loading time (ms) is listed in fig 5 for all combinations of event store and pruning algorithms. E.g. "Superseded: MongoDB" shows the mean loading time of 0.78 ms per aggregate after applying the superseded pruning algorithm on the EDN event store.

Table 3 contains a comparison of hydrating state via snapshots versus other pruning algorithms. The first row contains the mean loading time (ms) per aggregate for each event store using snapshotting. The following rows contain mean loading time (in ms and percentage) for each of the other six pruning algorithm, but as an offset from snapshotting. Positive numbers mean slower, negative means faster.

A last-minute bug was found in the command sourcing tests. As we did not have time to run all simulations again we ran one additional simulation and reported those numbers for command sourcing. This simulation contained 22 vessels. For validation, we took out measurements for no pruning state hydration to compare them against measurements from the main simulations. On average the new measurements diverged with 3.24%.

Pruning algorithm	MongoDB	Mysql	EDN files
No pruning	**10.57**	**81.03**	**57.66**
Superseded	(-92.62%) -9.79	(-79.40%) -64.34	(-96.64%) -55.72
Bounded	(-41.72%) -4.41	(-23.56%) -19.09	(-26.31%) -15.17
Probabilistic	(-96.12%) -10.16	(-83.98%) -68.05	(-99.79%) -57.54
Hierarchical	(-24.03%) -2.54	(-17.43%) -14.12	(-23.85%) -13.75
Snapshot	(-43.71%) -4.62	(-25.27%) -20.48	(-27.40%) -15.80
Command sourcing *	(+17.88%) +1.89	(+1.96%) +1.59	(+3.28%) +1.89

Table 5: Mean state hydration time (ms) per aggregate relative to no pruning. Negative numbers means faster, positive means slower. * From a limited set of tests (see 5.3.1)

5.3.2 Results

Looking at figure 5, we can see that aggregate state hydration on average ranged from 0.41 ms to 23.49 ms for MongoDB, from 12.98 ms to 97.07 ms for Mysql, and from 0.12 ms to 61.38 ms for EDN files. Across all event stores and pruning algorithms, the fastest average hydration time was 0.12 ms (Probabilistic on the EDN files event store) and the slowest was 82.62 ms (Command sourcing on Mysql), a factor of 688.5. If we exclude the algorithms that produce a different current state than originally (probabilistic and hierarchical), superseded had the lowest hydration time of 0.78 ms. Across all event stores, command sourcing had the slowest state hydration times.

By looking at table 5, we can see that most algorithms are faster than no pruning, except from command sourcing. Current literature often suggests using snapshots when the event log gets too large. Our results on the other hand show that most algorithms have similar performance as snapshotting. At least when the type of event store is taken into consideration. E.g EDN files.

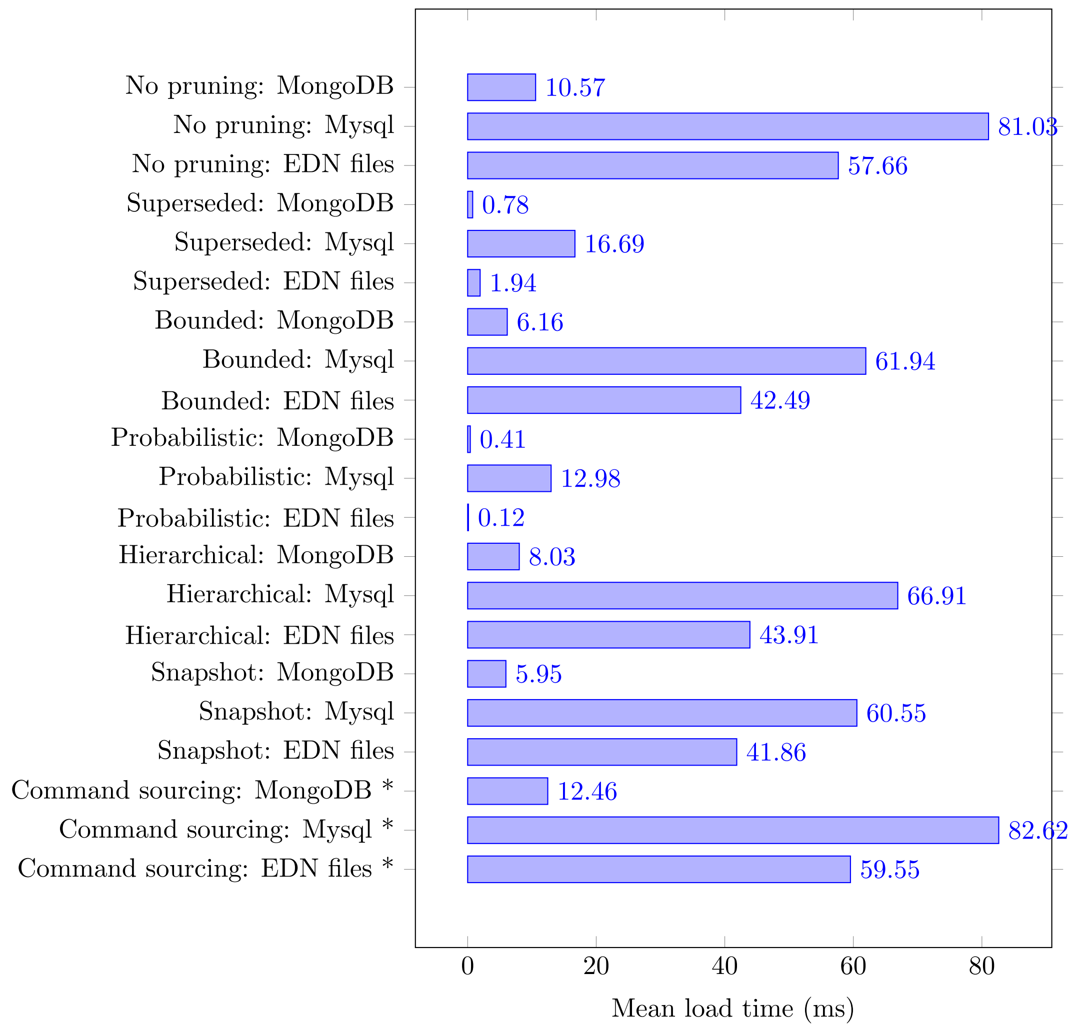


Figure 5: Mean load time per aggregate (ms) for all event store and pruning algorithm combinations. * From a limited set of tests (see 5.3.1)

5.4 Parameter sensitivity

The Parameter sensitivity tests were not executed as a part of the tests described earlier in this chapter. They were instead used as preliminary tests to find out how variations in inputs to the simulation model generator would affect the output model and state hydration. See also 4.3.2. These tests were executed without any pruning applied. From the tests, we extracted a number of events generated, mean loading time (ms), and event loading throughput (events per second).

In the first test series, we varied the AIS reporting interval from one report every second hour to one report every 1024 hours, doubling the report interval for each test. To isolate the utilization parameter, num vessels were set to 1, vessel age to 6 years, and utilization to 50%.

In the second test series, we varied the utilization interval from 0% to 100% with a 10% increase for each test. To isolate the utilization parameter, num vessels were set to 1 and vessel age to 6 years.

As the vessel age was set fixed to 6 years, events were generated for this period.

5.4.1 Data

Table 6 contains data for testing the AIS reporting interval. Each row represents a new test with a new AIS reporting interval. The first column "Interval" contains this interval. E.g. 1024h means that the vessel sends in one report every 1024 hours. "No. events" contains the number of events generated over the lifespan of the vessel (6 years). The next columns work in pairs for each event store. E.g., "MongoDB" contains the mean aggregate loading time on the MongoDB store. The next, "Events / s" column contains the event loading throughput per second for this test on MongoDB.

Table 7 contains data for testing utilization from 0% to 100%. This table has the same layout as table 6 but with utilization in the leftmost column instead of the AIS reporting interval.

5.4.2 Results

When the frequency of AIS reporting doubles the number of events nearly doubles as well. By setting the interval to 24h when running the main simulation we could expect around 2,500 additional events with between 11 ms and 74 ms of extra loading time across all stores for an average vessel. This is before any pruning and with the loading of other events subtracted (e.g. "vessel-created"). When reaching intervals of every 2h (at 26,521 events) mongoDB responded within 118.98 ms while both Mysql and EDN files started slowing down.

Interval	No. events	MongoDB (ms)	Events / s	Mysql (ms)	Events / s	EDN files (ms)	Events / s
1,024h	274	1.55	176,774	24.20	11,322	5.78	47,405
512h	323	1.74	185,632	25.79	12,524	7.08	45,621
256h	421	2.06	204,369	28.51	14,767	9.52	44,223
128h	645	3.02	213,576	36.55	17,647	15.43	41,802
64h	1,058	4.83	219,048	49.32	21,452	26.00	40,692
32h	1,874	8.55	219,181	71.29	26,287	47.07	39,813
16h	3,509	16.16	217,141	116.17	30,206	89.36	39,268
8h	6,800	30.42	223,537	210.19	32,352	174.51	38,966
4h	13,365	60.29	221,679	400.49	33,372	369.15	36,205
2h	26,521	118.98	222,903	760.79	34,860	715.10	37,087

Table 6: Parameter sensitivity for AIS reporting interval (no pruning applied). The interval is one report per n hours. Hydration time per event store is mean time in milliseconds for one vessel.

As utilization reached 100%, the number of generated events stopped at 28,644 with a total load time of 101.62 ms, 675.62 ms, and 523.63 ms for the event stores. Compared to the AIS reporting interval the utilization parameter did not affect the data model and load times nearly as much, even at max value.

Another aspect of testing the AIS report frequency is that it allowed us to investigate the maximum performance of the loading process. In table 6 there are three columns called "Events / s". They tell us how many events the system processed for the given parameter value. At the highest parameter values, 8h - 2h, we can see that the throughput started to stabilize at a max value of around 222,000 events / s for MongoDB, 33,000 events / s for Mysql, and 37,000 events / s for EDN files.

Utilization	No. events	MongoDB (ms)	Events / s	Mysql (ms)	Events / s	EDN files (ms)	Events / s
0%	2,194	10.36	211,776	81.64	26,874	58.38	37,581
10%	4,928	20.56	239,689	142.46	34,592	109.03	45,199
20%	7,640	29.55	258,545	205.66	37,149	157.03	48,653
30%	10,270	38.48	266,892	268.09	38,308	208.06	49,361
40%	12,744	47.16	270,229	323.91	39,344	253.11	50,350
50%	15,336	56.10	273,369	379.46	40,415	301.54	50,859
60%	18,098	65.39	276,770	444.85	40,683	354.86	51,000
70%	20,448	73.29	279,001	498.58	41,012	399.19	51,224
80%	23,234	80.36	289,124	563.60	41,224	427.62	54,333
90%	25,680	91.28	281,332	620.18	41,407	471.29	54,489
100%	28,644	101.62	281,874	675.62	42,397	523.63	54,703

Table 7: Parameter sensitivity for utilization (no pruning applied). Hydration time per event store is mean time in milliseconds for one vessel.

5.5 System implementation

Implementing the whole CQRS+ES system from scratch enabled us to have low-level control over most of the parts in the system. One example is that we were able to measure the performance of the parts we needed to measure to answer the research questions. However, this approach required us to make many design decisions to handle complexity and scope within a limited time frame. This section describes some of the challenges we experienced when implementing the CQRS+ES system.

Reducing complexity. To reduce complexity we divided up the system into smaller components (e.g., event-store, test-runner, and the command handler). This enabled us to isolate testing and bug fixing. In addition, the whole system was designed to run on one thread in one process, every step running in sequence.

Another important design decision was to represent both commands, events, and snapshots as events. This includes mechanisms for representation, hydration, persistence, and retrieval. A possible cost of this simplification is that we did not support the nuances of the command, event, and snapshot concepts. E.g., different metadata.

By reducing complexity, as described, we managed to keep control of the code-base and the testing process, but most likely at the cost of performance.

Preparing and running tests. To generate a correct simulation model we needed to generate events via the system. To do this we had to produce a given amount of commands to be executed. When trying recreate the whole population described in [24] we experienced issues with the feedback loop of

running tests as bugs needed to be resolved, and algorithms improved. Normally in a CQRS+ES application command execution can be expected to happen over time. Because of this, it was decided to test a reduced set of vessels each time. This improved the situation but not enough.

Some effort went into implementing parallel streams (one for the original stream and one for each pruning algorithm) but this approach was abandoned as we instead decided to focus on implementing support for a third event store. The solution to this problem was instead to back up all events before pruning and copy them back before the next test. If we had implemented the parallel streams instead, the pruning algorithms might have performed better as no delete operations would have been needed.

The test loop still took too long. To further improve command execution it was decided to implement support for an in-memory aggregate store. Instead of always going back to the event stores to build current state for an aggregate, every aggregate was kept in an in-memory map indexed by aggregate id. Running command execution on these aggregates reduced event store interaction to just persisting new events and commands.

Representing, persisting and retrieving events. Each event store has different characteristics. To support them all we needed to have a generic set of operations (e.g., to delete a stream, load a stream and delete a single event) and support for converting events to the storage format and back. When just supporting one event store, more effort could have been put into designing persistence interoperation according to the characteristics of the event store.

To enable fast retrieval of events from each store, their implementations were designed to utilize indices and other mechanisms to boost performance. E.g., for MongoDB we used binary to binary representation, co-loating all events for a single stream in one document, indexed by the aggregate UUID. For Mysql we added indices for finding and sorting all events for a given stream, however only with binary to text serialization and deserialization, and with blob storage for larger events (e.g., for snapshots). For the EDN files event store, we adopted a very simple approach of representing one stream as one file. To add events we serialized them before appending them to the end of the file. This could be done because of the single-threaded implementation of the system. To read events the process was reversed reading the whole file at once.

Pruning. The pruning functions had to utilize the event store operations to complete their work. Since the chosen solution was to not implement support for parallel streams, events had to be deleted, created, and inserted. At first, we implemented the simplest thing that could work: Delete whole streams and then insert them again as their pruned counterpart. This approach hurt the test feedback loop, and we had to refine the event store interface by adding support for deleting single events, deleting multiple events, backing up the event store, and rolling back the event store.

Projections. In our implementation, events are emitted to the projection event handlers in the same process as the rest of the application. This approach makes

sure all events are captured. It also ensures that we don't have any eventual consistency, a property normally found in CQRS+ES systems. This was done to reduce complexity and to have more control over the testing process.

Command sourcing. As command sourcing is a special state hydration process some measurements had to be taken to enable this to work. When dispatching commands, state is loaded via a "load-aggregate-es-" function that retrieves events from the event store. For command sourcing, a "load-aggregate-cs-" function instead reads commands from the event stores. Instead of generating the aggregate directly from specialized command handlers we run the commands as normal, but instead of persisting the produced events we return them to the "load-aggregate-cs-" function and apply them to the aggregate. When the first command is executed this aggregate is an empty map ({}). The next command receives this updated aggregate as a parameter. This approach ensures that we use the same state hydration functions for both command sourcing and event sourcing. By not building state directly, we reduced complexity and increased the quality of the test process, but at the cost of some overhead on processing time.

5.6 Summary

In this section, we have described the results from command execution, pruning, state hydration, parameter sensitivity analysis, and system implementation. For command execution, we have documented the expected throughput per second. For pruning, we have documented what the cost of a pruning mechanism might be and also the expected output in terms of number of events and stream sizes. For state hydration, we have documented hydration time across all pruning algorithms, and events stores, and compared them to state hydration without pruning. For parameter sensitivity, we have documented the effect of some model parameters on the generated data model in the form of expected state hydration time. In the end, we have documented some of the design decisions made when implementing a testable CQRS+ES system within a limited time frame.

6 Discussion

As described in the introduction chapter (1), calculating current state of an event stream gets harder as the stream grows. Current literature for the most part suggests using snapshots to handle this problem. Our motivation is to provide insight into this problem area to provide developer guidance.

Section 1.4 stated the goal of this thesis: "The thesis will provide insight into the performance of pruning algorithms combined with different storage solutions. The result will be to provide alternatives to snapshots for hydrating state in CQRS+ES within acceptable performance intervals.".

To achieve this goal we defined a research question in chapter 1.3: How can pruning and persistence of events affect state hydration and design of CQRS+ES systems? To answer this question a set of sub-questions were defined:

- 1.1 Which effect can pruning algorithms have on hydration performance?

- 1.2 Command sourcing can be classified as a form of log pruning. What is its effect on hydration performance?

- 1.3 To what extent does event granularity affect hydration performance and event retrieval?

- 1.4 Which design considerations related to state-building and persistence must be considered when implementing a CQRS+ES system?

To help answer these questions, we defined a methodology in section 4 that included developing and executing a simulation case on a CQRS+ES system. Simulation parameters were randomized and simulations were executed multiple times to be able to gather and aggregate measurements under different conditions. Results from these simulations were reported in section 5. This chapter will discuss these results to answer the research questions. The simulation case mainly focused on state hydration but we were also able to gain insight into command execution and the pruning process. By extracting knowledge from the simulation case we are able to contribute to the problem area as stated in section 1.4.

6.1 Event log pruning and state hydration (RQ 1.1)

In this section, we discuss event log pruning and state hydration via events. For state hydration via command sourcing and command execution see section 6.2.

6.1.1 Event log pruning

Our simulations reported a reduction in memory consumption and event count up to over 99% while still being able to calculate current state of some form (see 5.2). However, this pruning process has consequences. First of all, current state will differ based on which pruning algorithm is used. See more on this in section 6.1.2. Secondly, the pruning process takes time. In-process pruning required from under 1 second to over 105 seconds per aggregate, depending on algorithm and event store. Our simulation software mainly focused on hydration. I.e., the loading state and the effect of the pruning algorithms. For this reason it should be possible to further optimize the execution performance of the pruning algorithms. One example is the snapshot pruning algorithm. As it was implemented first it never got upgraded from deleting one and one event to batch deletion of events. Still, if we exclude this algorithm, the pruning took up to over 48 seconds for a single aggregate. However, several of the pruning algorithms are application-specific and can be tuned.

In any situation, pruning will take time and maybe leave the system in an inconsistent state while it is running. To run our simulation, 44,229 commands (on average) were executed in series to produce the correct events. In real production systems, there might not be a need to prune a similar amount of events at the same time. In addition, implementing pruning in the form of parallel streams might speed up the process by reducing the number of operations on the event stores (e.g., fewer delete operations). Selection and implementation of event store persistence will also affect performance. This will be further discussed in 6.4.

6.1.2 State hydration

The pruning algorithms in this paper can be divided into three categories: No pruning (unmodified event log and command sourcing), lossy pruning, and lossless pruning.

No pruning (including command sourcing) builds state without any modification of the event store. This enables us to handle the event log as immutable data by only adding new events, and never modifying or deleting existing data. Bug fixes and other changes to the streams must then be implemented as compensating events. Immutability enables easier distribution of data, scaling, and testing of the system. However, as shown earlier, the no pruning approaches has the longest state hydration times of all tested algorithms. In our vessel system, all events were designed with relatively high granularity and few commands produced more than one event. In a system with lower granularity and more events generated per command (for command sourcing) the no pruning approaches can be expected to perform better.

Lossless pruning modifies the event streams so that the current state is the same as when hydrating via the original commands. The main difference is that

we lose the ability to reconstruct intermediary states. In CQRS+ES, intermediary states can be used to reconstruct prior versions of an aggregate (e.g., for simulations), and to preserve user intent. The degree of reconstrucability was further explored in [7].

In this paper, superseded, bounded, and snapshots can be regarded as lossless pruning algorithms. These algorithms can be used as generic pruning algorithms in cases where prior states are not important.

For all pruning mechanisms, we expected snapshots to be the fastest as this is often used in current literature. However, measurements indicate that the superseded algorithm was by far the fastest of the lossless algorithms. It can be argued that this was caused by the design of the vessel management system as superseded is dependent on a domain-specific function to execute pruning. In the vessel system, we were able to remove a lot of events (all but the last of "position-reported" and "vessel-updated" events). In some systems, it might not be possible to remove any events with this algorithm. In any case, it should be a strong candidate when considering pruning mechanisms.

The bounded algorithm combines original events with snapshots. Tests showed that having one snapshot event in combination with a limited set of original events (99.97 events on average in our tests) had similar performance as full snapshots. This enables a speed increase while retaining a suitable granularity of newer events. E.g., in our vessel system we would keep the oldest "position-reported", "vessel-updated" and "period-added" events in a snapshot while keeping a more appropriate granularity of the newer ones. By tuning the number of events in each stream is possible to target a certain performance level on state hydration. E.g., always respond within 500 ms.

Lossy pruning modifies the event streams so that the new current state is not the same as the original current state. In our simulations probabilistic and hierarchical can be placed in this category. Measurements showed that probabilistic had the fastest overall hydration times. The hierarchical pruning algorithm also performed well. However, both the probabilistic and hierarchical algorithms are dependent on application-specific functions.

In the vessel management system, the probabilistic function was only focused on the future. All but the last of "position-reported" and "vessel-updated" events were removed. In addition, all vessel periods except current and future ones were also removed. Hierarchical has a similar implementation, removing more and more of the position reports as they age. As these algorithms remove most of the events in the streams, current state can easily be hydrated.

It is possible to utilize lossy pruning in cases where command execution does not require a full current state of an aggregate. If decisions are only made based on newer events there is no need to load older ones. An example can be drawn from the vessel system. If we want to calculate availability for a vessel we only need to investigate future periods.

In our CQRS+ES implementation of the pruning process, we manipulated the

original event streams (for both lossy and lossless pruning). Because of this, we lost reconstructability and user intent. This problem should be solved in real-world systems. A CQRS+ES system implemented with parallel streams (see 2.12) would be able to keep the original events while still boost performance by utilizing pruning.

No pruning, lossy pruning, and lossless pruning can be utilized in our vessel system. One strategy is to start off without pruning. If performance starts to dwindle over time we can add lossless pruning on parallel streams. This enables us to start off with a simple system and later on handle growth without changing code that is dependent on a fully hydrated aggregate. If performance continues to suffer or if we only need a partial state to make decisions, we can introduce lossy pruning for a limited set of commands.

Executing a command takes some time (see 5.1). In the vessel system, we utilized an in-memory aggregate cache instead of building state for every command. This enabled us to report average execution from 1.27 ms to 45.33 ms across all events stores. In these measurements, we consider the time spent on in-memory lookup to be negligible. A reasonable response time can be set to 1 second (defined as a deadline for "finishing user-requested operations" in [13] page 161). If we add measurements of command execution to our state hydration measurements (0.12 ms - 91.07 ms. See 5.3) we are still well within this limit (1.39 ms to 196.95 ms). A consequence of this is that all state hydration algorithms can be applied to a CQRS+ES system with similar characteristics as our vessel system.

At some point, each pruning algorithm will fail to provide fast enough responses. Our parameter sensitivity tests for the AIS report interval provide ut with indicators for when this can be expected to happen for each event store. See section 5.4. Throughput seemed to peak at 222,000 events per second for MongoDB, 33,000 events per second for Mysql, and 37,000 events per second for EDN files. I.e., MongoDB should be able to process an event stream of 222,000 events minus command execution time within the 1-second limit defined earlier.

6.2 Command sourcing and state hydration (RQ 1.2)

A requirement for using command sourcing is that it must be possible to execute a series of commands multiple times with the same result (current state). This can be achieved by persisting command parameters. In addition, each command handler should ideally only be dependent on the parameters passed to them. If they need to interact with mutable resources (clocks, storage, or external resources), those values can be persisted along with the command data.

Our test of command sourcing shows that it had a similar performance as no pruning, across all event stores. However, command sourcing has three phases: The first phase loads all commands from the event store. The second phase generates events. The third phase hydrate state. In the vessel system, we have fewer commands than events. This means that command sourcing can spend

less time on loading streams but some additional time on generating events. The hydration step is expected to be the same. If we had measured each step we should have been able to see that the initial stream loading was faster than no pruning stream loading. If the vessel system on average produced a lot more events than commands this effect should be observable in the total state hydration time as well.

It can be argued that command sourcing provides a viable alternative to event sourcing. Based on our measurements, command sourcing had a similar performance as event sourcing with no pruning. Compared to the pruning algorithms, command sourcing performed worse. However, performance can still be considered to be within an acceptable range for command execution in the vessel management system (12.46 ms to 82.62 ms). An additional argument in the case for command sourcing can be found by looking at the optimization of command execution. In our simulations, we measured the performance of command execution in the form of commands executed per second (see 5.1). This included the time it took to apply generated events on an in-memory (i.e., cached) aggregate, and then persist the produced events and the command itself. By combining in-memory aggregates with command sourcing on an append-only event store (EDN files) we were able to execute 789.80 commands per second using a single thread on mediocre hardware. It must be noted that this is also a viable approach for the other state hydration mechanisms.

When the deterministic property of command sourcing has been addressed (i.e., making sure command callers only pass static parameters such as new UUIDs and timestamps), the rest of the implementation is straight forward. Persisting data is always done by appending new commands to the event store. Building state is always done by reading the commands and executing them again. As we just described, in our simulation software we chose to add an intermediary step between command execution and current state to enable comparison of algorithms. Our command handler produces events that are then applied. It is possible to skip this step and build state directly. For command sourcing, the pruning step is not relevant. This reduces complexity and overall processing time.

Command sourcing has intrinsic support for parallel realities and time travel. With parallel realities it is possible to explore alternative execution paths and data models based on the same set of commands, but with different execution. This can be done for both prior states or current state. Two examples of this can be drawn from the vessel management system. It might not have been the best idea to put position reports inside the vessel aggregate as it grows steadily over time. By changing the command handler implementation it would be possible to put position reports in a different aggregate to test the effect on system performance. Time travel in the vessel management system would be possible by executing commands up to the point in time that a user or a developer wants to investigate.

6.3 Event granularity (RQ 1.3)

Event granularity is related to the size and quantity of events in an event stream. To represent the same current state, few events result in low granularity. Many events result in high granularity. Granularity affects to what degree intermediary states can be reproduced. With snapshotting we have one event with low granularity (no intermediary states). With no pruning, we have many events with high granularity (all intermediary states). Every other pruning algorithm in between (except command sourcing) can be placed on a continuum between no pruning and snapshotting.

In our implementation of the CQRS+ES system, we chose to implement snapshots as a single large event as this allowed us to re-use existing code and compare measurements. Our tests (see 5.2 and 5.3) showed an improvement in hydration time of 25% to 44% and a reduction of memory consumption around 29% when using snapshotting. This comes at a cost of losing reconstructability and increased complexity.

For some data models, snapshotting can be viewed as an extreme version of superseded pruning. This is the case when the shape of current state only contains a single value (e.g., the latest version of a value such as last updated). On the other extreme, all data from all the original events (e.g., as a list of periods) can still be present in the aggregate. Our test data model contains a mix of both. For each manipulation of a vessel, we calculated a last updated property and we kept the last reported position in its own property. To build a complete vessel history we always kept all reported periods in a list. However, the snapshot version will in most cases require less memory consumption as there is less metadata to process.

Our implementation of the vessel management system manipulated the original event streams to execute pruning. A consequence of this is that it is not capable of reconstructing intermediary states after pruning. We did implement a backup and recovery function, but that was only a tool to improve test execution. Let's say "add-period" and "remove-period" commands are executed on a vessel over its lifetime to manipulate operational history. Then a new application requirement comes in where users should be able to see the operational history as of last year. With pruning, this is impossible unless it can be achieved through event payload data. The granularity is too low and we might have lost data.

We have a similar situation with user intent. As an example, a "vessel-name-updated" can be defined to represent name changes, or we could have a more generic "vessel-updated" event. In this case, it might be possible to derive user intent by inspecting the payload of an event (e.g., the user only changed the vessel name in the generic "vessel-updated" event). A consequence of these examples is that the application designer is responsible for deciding the future value of event data. Both in terms of event design and pruning implementation.

6.4 Design considerations (RQ 1.4)

In this section, we have divided the discussion into four parts. Each part contains a discussion of design decisions based on implementing the CQRS+ES test system and running tests on it. We start by describing a simple command-sourced system for smaller projects or microservices. Thereafter we describe a larger CQRS+ES system utilizing pruning. Following this, we discuss design considerations related to projections and event stores.

6.4.1 A command-sourced CQRS system

A simple versus intertwined system is easier to understand, extend and manage. Immutability makes debugging easier and increases scalability. Our implementation of the CQRS+ES system focused on this kind of simplicity to enhance testability, and to be able to complete a working system within a limited time frame. Except for running every process step in sequence we achieved to implement a simple CQRS+ES with acceptable performance, even when running on a single thread.

The vessel management system is able to execute 789.80 commands per second when using the EDN file store in combination with in-memory cached aggregates (See 5.1). This time included persistence of the command and at least one event. In such a system all aggregates must either be loaded on startup or put into memory the first time it is requested. We measured this process to take between 12.46 ms and 82.62 ms in the vessel management system (see 5.3) when processing 2,316.64 events. In addition, aggregate state must be continuously updated as new commands are executed.

Much of the speed of command execution in our simulation system can be attributed to EDN file store implementation. Appending events to the end of a file takes little time. Our implementation of the EDN file store used the Clojure spit function to write to files. This makes sure every write is flushed to disk before continuing, making sure we really persisted data instead of keeping it in memory. In addition, we convert data from binary to text before writing. Performance might be further improved by switching to stream-oriented file APIs (java.io.Writer) and binary persistence.

In command-sourced systems, it is only the command handlers that can write to the event store. By placing them on a single thread with exclusive write access we should be able to reduce complexity and enhance consistency guarantees. Limiting write access to one thread can at first glance seem to result in low performance. However, this thread will only deal with command execution (writing). Other threads and data models (projections) can be scaled up to meet requirements for appropriate reading performance.

This system can't be expected to be appropriate in all situations, even if we have utilized parallel realities as described in 6.2 to improve performance. Memory

requirements, number of commands per aggregate, or command payload size might render the system too slow for practical purposes. A solution to this problem is to switch to event sourcing with pruning. Switching is a three-step process. Firstly, events produced by the command execution process must be persisted for the whole command history. Then state hydration must be done by loading events from the event store instead of command sourcing. In the end, pruning can be introduced if needed. In the vessel management system, we had support for this approach. We generated a set of commands to build the target data model. These commands were both used to test command sourcing an to produce events to be pruned and tested (see 4.3).

6.4.2 A CQRS system with pruning

The approach described in 6.4.1 related to command processing is still valid for an event-sourced system with pruning. However, loading each aggregate into memory on startup might not be feasible. Loading aggregates into memory when they are needed and dismissing them after a while reduces the memory footprint of the system. This way we don't have to interact with aggregates that are not used often.

In 5.3, we showed that the vessel management system could utilize state hydration without pruning. This should be transferrable to systems with similar characteristics. By starting with the no pruning approach we can reduce the complexity of the system implementation. As state hydration starts to slow down it is possible to introduce parallel event streams. Parallel event streams enable us to keep the original events and still retain properties such as immutability, intermediate state reconstruction, and user intent. It also enables us to introduce a varied set of pruning mechanisms that can be replaced or upgraded throughout the lifespan of the system. The original events can be kept in their own streams and pruned streams can be created from traversing these streams. Care must be taken to make sure decision making on original, and pruned events are based on the same facts. One solution to this problem is to utilize in-process pruning. Tests on the vessel management system showed us that on average, superseded pruning added 20.61 ms per event with in-process pruning (see 5.2) when using the EDN event store.

By tailoring the application domain pruning functions for the superseded, probabilistic and hierarchical algorithms it should be possible to obtain performance metrics far beyond what we experienced in the simulation CQRS+ES system. A disadvantage of the domain-aware functions is that this must be implemented especially for each system. To be able to do this successfully we must have intimate knowledge of events and aggregates. On the other hand, implementing the more generic snapshot and bounded pruning algorithms enables us to use it on many different aggregate types.

In our tests, the superseded algorithm performed best of the lossless algorithms. The performance of this algorithm was tied to the structure of each aggregate. We removed all but the last of "position-reported" and "vessel-updated" events,

a relatively high percentage of the events in each aggregate. Because of this, performance must be expected to vary between systems. However, this algorithm is simple to both implement and estimate performance on. In addition, pruning time should be relatively constant. When a new event arrives, the pruning algorithm only needs to find the preceding event of the same type (if it represents superseded state) and remove it.

6.4.3 Projections and command execution

When designing a CQRS+ES system the command handlers are used to write to the system. Projections are used to read from the system. In our simulation, we utilized projections to be able to execute queries across multiple aggregates. We only supported one type of query: Find all vessels or fixtures. Projections are used to build read models. When executing commands based on input from the projections it must be taken into considerations that they are eventually consistent. By passing in all parameters the command handlers needs (including aggregate ids for creating new aggregates), we can remove all database interaction from the command handlers except for loading state via commands or events. In addition, it enables us to support command sourcing because immutable parameters can be persisted and used to achieve deterministic execution.

In 6.4.1, and 6.4.2, we discussed design consideration for two types of CQRS+ES systems. Both of these systems rely on using projections for all read operations. This reduces the amount of work needed to be done by the command handlers when comparing to a system that uses one data model for both reading and writing.

Projections can be stored separately from the event store, and they can be implemented with any persistence mechanism. This is a great advantage as we can adopt the correct technology to represent a read model. In the vessel management system, we used Mysql to store projections. This database was colocated with the Mysql event store for practical purposes. However, the same projections were used when testing all three event stores.

We have already shown how a projection can be used to fetch a list of all vessels. The versatility of projections can be illustrated by adding two new requirements to the vessel management system. In the first example, we want to add system-wide search capabilities. This can be done by adding projection event handlers that update an indexing service (i.e., a projection) such as Solr. In the second example, we want to build a user interface for managing vessels. By adding new projection event handlers, we can precompile data structures that contain all data needed for rendering a specific UI element.

6.4.4 Implementing event stores

In our CQRS+ES implementation, we added support for three event stores. Each of these event stores has strengths and weaknesses to consider. Selecting one over the other requires knowledge of expected data structures and usage. Supported database operations and performance must be considered for both retrieval and modification of events, and streams. To shed light on this, we will describe some of the issues we faced when implementing the simulation software.

For MongoDB we used binary to binary persistence, keeping all events for one stream in one document. This enabled easy retrieval of all events with one operation, and minimal processing between data formats. However, as the event log grew we reached the maximum document size for MongoDB (16 MB). In addition, insert speed decreased as more and more events were added to a document. To support event streams over 16 MB would require an extension of the event store. E.g., by including a ”next-document” property in each document. When this property is set the next document in the chain must be retrieved as well, further increasing hydration time.

For Mysql, we used binary to text serialization and kept all events in one table. To enable fast sorting and retrieval of events we introduced two indices on the event table. The first one for retrieving all events for a stream id. The second one for sorting (including the event timestamp). In this situation, Mysql can use the first index to retrieve all events for a stream. If we add 'order by t asc' when a stream is retrieved, Mysql can utilize both indices to serve us an already sorted event stream. A downside of these indices is that they must be managed. This adds overhead when inserting events. For this store, we did not implement any binary to binary support, which could have improved performance. In addition, limits on row size forced us to persist the larger event payloads (e.g., for snapshots and bounded pruning) outside of the table page files.

For EDN files we used binary to text serialization, placing one stream in one file. Binary to binary data conversion could also have improved speed for this event store. Flat file storage solutions are seemingly simple and easy to implement. However, the main drawback of our implementation is that there is no tool support and consistency guarantees. Measurements showed that our append-only approached yielded the best pruning performance and similar hydration performance as MongoDB, even with binary to text conversion. This might not be the case with a more elaborate implementation. In section 6.4.1, we suggested using one single thread for managing writes to increase consistency and reduce complexity. A similar approach can be found in the Datomic DBMS where a single process manages writing and other processes manage reading.

Comparing a flat-file EDN approach with Mysql and MongoDB is not completely fair, but it helps to illustrate the available toolset when implementing an event store. E.g., MongoDB and Mysql are full-fledged database management systems that have data quality guarantees (e.g., ACID and BASE). In addition, both MongoDB and Mysql performance can be further tuned (e.g., by adjusting the

use of in-memory caches). The EDN event store for the vessel management system is completely without any guarantees of any form. In addition, we did not test other database systems such as graph databases, column databases, and key-value stores. As shown in the vessel management system we were able to utilize multiple storage mechanisms because of the simple data structures in CQRS+ES. In practice, it is possible to start with one persistence type and then change it later on if system characteristics or requirements change.

6.5 Reproducibility

With some effort and access to similar hardware as we ran our tests on, it should be possible to produce similar results as reported in this paper. The source code of our simulation software is available online for download and review. In section 4.3.4 information about simulation hardware can be found. Appendix A contains instructions for how to download, install and run simulations. It also contains information about the software used and installation instructions for each event store. Appendix B documents randomization and static parameters used to generate the commands and events that were used in the simulations. Actual randomized parameters and reported measurements are also available online for each of the tests we ran. See appendix C.

6.6 Limitations

The findings in this paper were based on implementing a complete CQRS+ES system including support for three different persistence mechanisms. As in any software implementation, and algorithm there is room for optimizations. Some of these we have already addressed before. E.g., binary to text serialization and combining multiple delete operations into one operation. Also, some of the implemented pruning methods could be improved by tailoring either the algorithm itself or the event store interface to the characteristics of each algorithm. Earlier we mentioned the use of parallel streams. This concept was not tested in our simulation. In addition we ran all tests in sequence on an isolated system to have better control over the testing process and measurements. We did not investigate the consequences of concurrently executing commands and pruning.

The simulation data model was developed based on population estimates and not actual data. Testing was only executed on a subset of this population. Also, the requirements for the application were derived from prior experience in the field, but not from an actual working system. The main purpose of this system was to enable discussions and testing of data structures in relation to pruning. E.g., superseded values, lists, and aggregate sizes. We do however expect to find similar data structures in other software systems.

The superseded, probabilistic and hierarchical pruning algorithms are not generic must be implemented based on the target aggregates. This means that measurements will differ from implementation to implementation.

It must be noted that the goal of this paper was to investigate the potential effect of pruning and state hydration algorithms to guide other developers when designing CQRS+ES systems. The goal was not to produce absolute measurements.

7 Conclusion

This paper aimed to find out how pruning and persistence of events affect state hydration, and the design of CQRS+ES systems. To enable discussion we defined a methodology that included designing, implementing, and testing a CQRS+ES system. This system was developed with support for event sourcing, command sourcing, and a set of lossy and lossless pruning algorithms. In addition, we added support for the persistence and retrieval of events via MongoDB, Mysql, and an EDN-based flat-file store.

We have shown how pruning algorithms can reduce the size of the event streams to less than 1% of their original size. This comes at the cost of increased complexity, and reduced reconstructability. In the case of lossy pruning, we also lose some details of current state, which in turn might require different implementations of event handlers. A different, but viable approach is to not persist events at all, and instead produce them through command sourcing.

The pruning and state hydration algorithms can be divided into three categories: No pruning (command sourcing, and unmodified event logs), lossless pruning (superseded, bounded, and snapshots), and lossy pruning (probabilistic, and hierarchical). The no pruning state hydration algorithms are less complex to implement, and do not break immutability, but has the lowest performance. The lossless pruning algorithms remove intermediary reconstructable states but produce the same current state as the no pruning algorithms. The lossy algorithms produce a different current state than the no pruning algorithms. This means they can't be used in cases where a full aggregate is needed. The pruning algorithms can also be divided into generic, and application domain aware functions. Generic pruning algorithms such as snapshots can be utilized across different aggregate types, but generally had a poorer performance in our tests. The application domain aware pruning algorithms are customized for each aggregate, and application. In our tests, these algorithms were the fastest. E.g., we were able to hydrate current state up to 480.5 times faster with probabilistic pruning, and 27.72 times faster with probabilistic pruning than no pruning. To avoid losing data with lossless and lossy pruning, a mechanism supporting parallel streams must be implemented. This adds complexity. As event streams grow, the throughput of the system must be expected to dwindle. Measurement on the vessel management system showed that the supported event store was able to load around 222,000 events per second for MongoDB, 33,000 events per second for Mysql, and 37,000 events per second for EDN files. All on mediocre hardware.

For event granularity, we have shown that streams of low granularity perform better than streams of high granularity. We were able to measure a reduction in state hydration time of 25% to 44%, and a reduction in memory requirement around 29%. The cost of this approach is the loss of reconstructability and user intent.

We have also shown that command sourcing is a viable alternative to event

sourcing. By re-executing commands for every hydration in combination with in-memory cached aggregates, we were able to achieve similar performance as a CQRS+ES system without pruning. At the same time, we were able to support important properties such as immutability, time travel, extensibility, and alternative execution paths.

We went on by showing a design and implementation strategy for a CQRS+ES system. A simple command-sourced system is firstly designed by utilizing in-memory cached aggregates, and a single thread for managing command execution. When the system becomes too slow it is possible to generate events for all commands, persist them, and continue as an event-sourced system (I.e., with persisted instead of in-memory events). This includes implementing support for parallel streams and pruning algorithms. By starting with generic lossless pruning algorithms the event handlers can stay the same as for command sourcing. If performance continues to be a problem, lossy domain aware pruning mechanisms can be employed. A key factor of this approach is the relief given by the projections. They enable command sourcing by handling all read queries and passing static parameters to the command handlers.

In the end, we showed some implications and pitfalls of implementing event stores. As the data structures of a CQRS+ES system are simple, we were able to implement persistence via three different database systems utilizing both text and binary storage formats.

Our findings demonstrate that there is a range of alternative design options available compared to what can be found in current literature on CQRS+ES systems. By utilizing command sourcing, and pruning algorithms that are aware of the application domain we can tune the system towards a varied set of application requirements. There is a significance to this. It should now be easier for developers to employ a CQRS+ES architecture in areas currently dominated by other system architectures.

As stated earlier, our findings are based on some limitations. Our simulation was based on model generation, and not a real vessel management system. We also implemented the whole CQRS+ES system including pruning, and state hydration algorithms from scratch. This implementation can be improved further. In addition, the domain aware pruning algorithms will result in different performance metrics on other systems with different data model characteristics. For the event stores, we compared persistence mechanisms with different capabilities, representation, and characteristics.

7.1 Future work

In this paper, we have shown that command sourcing has a lot of interesting properties. However, the topic is not much covered by current literature. This means that properties such as scalability (e.g., concurrent binary persistence), variability (e.g., extensibility, and alternative execution paths), and maintainability (e.g., tooling, and compensation) should be explored further. We also

need more research on real production systems for command sourcing.

In our discussion of more complex CQRS+ES systems we suggested some design guidelines (e.g., for state hydration, concurrency, immutability, reconstructability, command execution, and persistence). The consequences of these guidelines should be explored in real production systems in combination with the more promising pruning algorithms, such as superseded and probabilistic pruning.

For our simulation software, we implemented three types of event stores. A document store, a relational database, and a flat-file store. In addition to doing more work on binary persistence we also need to explore other persistence mechanisms such as graph databases, column databases and, key-value databases to understand how they can contribute to the development of CQRS+ES systems.

List of Figures

1 Command handling 11

2 Building state from an event stream 13

3 System partitioning and workflow 26

4 Testing process 35

5 Mean load times per aggregate 43

List of Tables

1 Event store layers and operations 10

2 Mean command execution time 39

3 Mean aggregate pruning time 40

4 Events after pruning 41

5 State hydration relative to no pruning 42

6 Parameter sensitivity for AIS reporting interval 45

7 Parameter sensitivity for utilization 46

Listings

1 Event data structure 11

2 Command data structure 13

3 Original events 14

4 Pruned events (from listing 3) 15

5 Vessel aggregate after combining events from listing 3 or 4. 15

6 System entities. 22

7 Command data structure 27

8	A command handler for the vessel aggregate	27
9	A state building function for the vessel aggregate	28
10	Polymorphic persistence support	29
11	Mysql event and command tables.	30
12	Projections stored in Mysql.	30
13	MongoDB event/command collections and example event stream	31
14	Flat file content and file names	32

References

[1] Danish Maritime Autority. Ais data. `https://www.dma.dk/SikkerhedTilSoes/Sejladsinformation/AIS/Sider/default.aspx`. [Online; accessed 18-november-2020].

[2] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. Exploring cqrs and event sourcing: A journey into high scalability, availability, and maintainability with windows azure. page 270, 2013.

[3] Udi Dahan. Clarified cqrs. `http://udidahan.com/2009/12/09/clarified-cqrs/`, 2009. [Online; accessed 18-May-2019].

[4] Andrzej et al. Debski. In search for a scalable & reactive architecture of a cloud application: Cqrs and event sourcing case study. *IEEE Software*, (99), 2017.

[5] Benjamin Erb and Frank Kargl. A conceptual model for event-sourced graph computing. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 352–355. ACM, 2015.

[6] Benjamin Erb, Dominik Meißner, Gerhard Habiger, Jakob Pietron, and Frank Kargl. Consistent retrospective snapshots in distributed event-sourced systems. In *2017 International Conference on Networked Systems (NetSys)*, pages 1–8. IEEE, 2017.

[7] Benjamin Erb, Dominik Meißner, Ferdinand Ogger, and Frank Kargl. Log pruning in distributed event-sourced systems. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 230–233. ACM, 2018.

[8] Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 78–87. ACM, 2017.

[9] Ehren Thomas Eschmann. Maintaining parallel realities in cqrs and event sourcing, 2017.

[10] Martin Fowler. Retroactive event. `https://martinfowler.com/eaaDev/RetroactiveEvent.html`, 2005. [Online; accessed 11-november-2020].

[11] Martin Fowler. Event sourcing. `https://martinfowler.com/eaaDev/EventSourcing.html`, 2015. [Online; accessed 9-Nov-2020].

[12] JPJ Guelen. Informed cqrs design with continuous performance testing. Master's thesis, 2015.

[13] Jeff Johnson. *Designing with the mind in mind*. Morgan Kaufmann, 2010.

[14] Jaap Kabbedijk, Slinger Jansen, and Sjaak Brinkkemper. 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*, page 2. ACM, 2012.

[15] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. Retro-λ: An event-sourced platform for serverless applications with retroactive computing support. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, pages 76–87, 2018.

[16] Bertrand Meyer. *Object-oriented software construction - 2nd ed.* Prentice Hall, 1997.

[17] Michael Müller. Enabling retroactive computing through event sourcing. Universität Ulm, 2016.

[18] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[19] Michiel Overeem, Marten Spoor, and Slinger Jansen. The dark side of event sourcing: Managing data conversion. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 193–204. IEEE, 2017.

[20] Johan Rothsberg. Evaluation of using nosql databases in an event sourcing system, 2015.

[21] Aleksey Shipilev. Nanotrusting the nanotime. `https://shipilev.net/blog/2014/nanotrusting-nanotime/`. [Online; accessed 28-november-2020].

[22] MJ Spoor. Efficiently handling data schema changes in an event sourced system. Master's thesis, 2016.

[23] Ben Stopford. Designing event driven systems. pages 18, 24, 37, 2018.

[24] Morten Tvedte and Alexander Sterud. Obtaining contracts in the north sea osv market. pages 19–20, 2016.

[25] Brian Ye. An evaluation on using coarse-grained events in an event sourcing context and its effects compared to fine-grained events, 2017.

[26] Greg Young. Cqrs documents by greg young. `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`, 2010. [Online; accessed 9-Nov-2020].

[27] Greg Young. *Versioning in an Event Sourced System.* Leanpub, 2017.

[28] Yifan Zhong, Wei Li, and Jing Wang. Using event sourcing and cqrs to build a high performance point trading system. In *Proceedings of the 2019 5th International Conference on E-Business and Applications*, pages 16–19, 2019.

Appendices

A Running experiments

This section contains instructions on how to run the same experiments as used in this paper. After running the testsrunner and analytics software, output files will be available under /mnt/data for furter review:

- testrunner: Results from executing the tests. These are in EDN format and saved as timestamp.edn.
- analytis: Output from the analytics module. These are generated from parsing the files in the testrunner directory.

Under the same directory there will also be an event-store directory containing the text based EDN database. Test data for MySQL and Mongodb will be placed in their default locations.

Experiment source code can be pulled from Github at https://github.com/marcusba/cqrs-test/archive/31ef158d03ecf64d34960aa4e97bf0a695ecf37a.zip with last commit hash of 31ef158d03ecf64d34960aa4e97bf0a695ecf37a. This version was used to run simulations. To check current sha1 has run this command in the project (git) directory: git rev-parse HEAD. The install procedures were last tested as of 2020-11-07.

Installing and running the simulation:

```
#Installation of software from clean and updated Ubuntu 20.04:
#start from home dir
cd

#Set up Clojure
#=====================================================
#Java
sudo apt install openjdk-8-jdk

#Get the experiment code
git clone https://github.com/marcusba/cqrs-test.git

#Clojure
sudo apt install rlwrap
curl -O https://download.clojure.org/install/linux-install-1.10.1.536.sh
chmod +x linux-install-1.10.1.536.sh
sudo ./linux-install-1.10.1.536.sh

#Leiningen
```

```
wget https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein
chmod a+x lein
./lein
sudo mv lein /usr/bin

#Set up Mysql
#==================================================
#mysql community server Ver 8.0.22-0ubuntu0.20.04.2 for Linux on x86_64 ((Ubuntu))
sudo apt install mysql-server

#set up timezone for mysql
mysql_tzinfo_to_sql /usr/share/zoneinfo | sudo mysql -u root mysql
#add timezone /etc/mysql/my.cnf:
printf "[mysqld]\ndefault-time-zone='Europe/Oslo'" | sudo tee -a /etc/mysql/my.cnf

#set up cqrs database
sudo mysql -u root
create database event_store character set utf8 collate utf8_danish_ci;
use event_store;
create table event (
 t bigint not null,
 s char(36) not null,
 n varchar(50) not null,
 e int not null,
 p varchar(21000) default null,
 pb mediumtext default null
) ENGINE=MyISAM;

create table event_backup like event;
create table command like event;

create index tse on event (t,s,e);
create index se on event (s,e);
create index tse on event_backup (t,s,e);
create index se on event_backup (s,e);
create index tse on command (t,s,e);
create index se on command (s,e);

#Add projections
CREATE TABLE `all_vessels` (
  `aggregate` char(36) COLLATE utf8_danish_ci NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

CREATE TABLE `all_fixtures` (
  `aggregate` char(36) COLLATE utf8_danish_ci NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

create user 'event_store'@'localhost' identified by 'password';
grant all privileges on *.* TO 'event_store'@'localhost';
flush privileges;
```

```
exit

#Set up mongodb
#=====================================================
#mongodb community edition v 4.4.1 git version ad91a93a5a31e175f5cbf8c69561e788bbc55ce1
wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -
sudo apt-get install gnupg
wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org
/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list
sudo apt-get update
sudo apt-get install -y mongodb-org
echo "mongodb-org hold" | sudo dpkg --set-selections
echo "mongodb-org-server hold" | sudo dpkg --set-selections
echo "mongodb-org-shell hold" | sudo dpkg --set-selections
echo "mongodb-org-mongos hold" | sudo dpkg --set-selections
echo "mongodb-org-tools hold" | sudo dpkg --set-selections
sudo systemctl start mongod
sudo systemctl daemon-reload
sudo systemctl status mongod
sudo systemctl stop mongod
sudo systemctl restart mongod
#set up cqrs databse
mongo
use event-store;
db.createCollection('event');
db.createCollection('event_backup');
db.createCollection('command');
db.createUser({user: "event-store", pwd: "password", roles: [{role: "readWrite",
db: "event-store"}]});
exit

#Prepare experiment software
#=====================================================
#Install components to ~/.m2/repository
cd cqrs-test
cd util
lein install
cd ../event-store
lein install
cd ../command
lein install
cd ../pruning
lein install

#Run experiment
#=====================================================
#Edit src/core.clj test-config data structure to modify parameters for the tests
#Output placed under /mnt/data/testrunner
cd
```

```
cd cqrs-test/testrunner
lein repl
(execute! test-config)
exit
```

B Simulation parameters

This appendix contains a list of paramteres used to configure event stores and orchestrate the simulation process.

Static and random simulation parameters:

```
(def num-vessels (rand-range-int 12 31))

(def test-config {
   ;simulation data model generation
   :num-vessels num-vessels ; num vessels (population = 120 - 313 avg 6-9 y)
   :min-vessel-age 0
   :max-vessel-age 11
   :preferred-vessels [40 1.3] ; 40% of vessels are preferred with a weight of 1.3.
   :min-vessel-utilization 0.31
   :max-vessel-utilization 0.72
   :min-contract-length 3
   :max-contract-length 30 ; sport market <= 30 days
   :ais-event-resolution 24 ; hours between each report

   ;dates
   :start-date-time (t/zoned-date-time 2021 1 1) ; simulation starts
   :end-date-time (t/zoned-date-time 2031 12 31) ; simulation ends

   ;pruning
   :bounded-buffer 100 ; keep last 100 events. snapshot of the rest
   :hierarchical-windows [30 59 2 60 99999 4] ; day 30-59 drop 50%, >60 days drop 75%

   ;event-storage
   :storage-mysql {:type :mysql
                    :name "event_store"
                    :user "event_store"
                    :password "password"
                    :host "localhost"
                    :port 3306}

   :storage-multi-file-edn {:type :multi-file-edn
                             :directory "/mnt/data/event-store"}

   :storage-mongodb {:type :mongodb
             :name "event-store"
             :user "event-store"
             :password "password"
             :host "localhost"
             :port 27017}

   ;projection storage - same for all event stores
   :projections-storage {:type :mysql
```

```
                        :name "event_store"
                        :user "event_store"
                        :password "password"
                        :host "localhost"
                          :port 3306}

   ;test orchestration
   :test-iterations 1000 ;how many times each aggregate will loaded for each test
   :cooldown-time 300 ;sec delay before each test set (per pruning)
   :test-output-dir "/mnt/data/testrunner"
})
```

C Test results

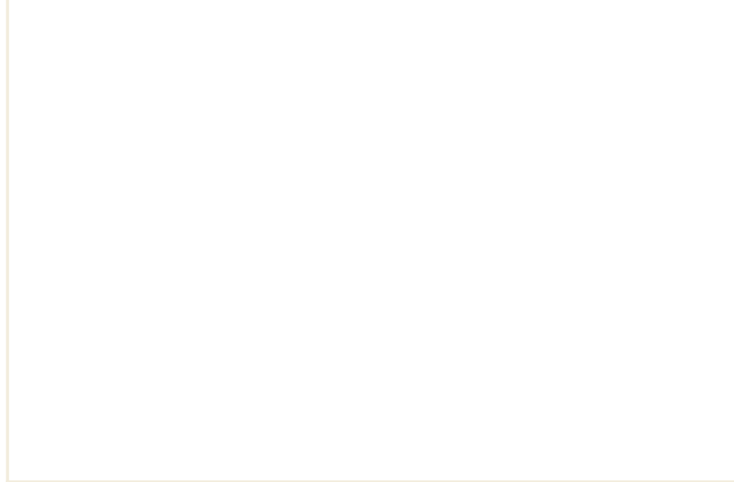
The simulation in this paper was repeated five times. The output of this simulation, generated simulation parameters and measurements, can be accessed online.

URL: https://github.com/marcusba/cqrs-test/tree/master/test-results.

There are two file types in EDN format:

n-params.edn contains parameters for test n.

n.edn contains measurements for test n.



NTNU
Norwegian University of
Science and Technology