Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Evaluation of using NoSQL databases in an event sourcing system

by

Johan Rothsberg

LIU-IDA/LITH-EX-A—15/068—SE

2015-11-24

Linköpings universitet

Linköping University
Department of Computer and Information Science

Final thesis

Evaluation of using NoSQL databases in an event sourcing system

by

Johan Rothsberg

LIU-IDA/LITH-EX-A—15/068—SE

2015-11-24

Supervisor: Valentina Ivanova
Examiner: Patrick Lambrix

Abstract

An event store is a database for storing events in an event sourcing system. Instead of storing the current state, a very common way to persist data, an event sourcing system captures all changes to an application state as a sequence of events. Usually the event store is a relational database. Relational databases have several drawbacks and therefore NoSQL databases have been developed. The purpose of this thesis is to explore the possibility of using a NoSQL database in an event sourcing system. We will see how data is stored in an event store and then evaluate different solutions to find a suitable database. The graph database Neo4j was selected to be further investigated and a Neo4j event store has been implemented. At last the implemented solution is evaluated against the existing event store that uses a relational database. The conclusion of this thesis is that event store data could easily be modeled in Neo4j but some queries became complex to implement. The performance tests showed us that the implemented event store had poorer performance than the existing one using a relational database.

Contents

List of Figures

List of Tables

List of source codes

Chapter 1

Introduction

1.1 The company

This master thesis was conducted at Upptec in Malmö, Sweden. Upptec supplies a Software as a service (SaaS) to Swedish insurance companies for valuation of stolen, broken or lost property.

1.2 Motivation

The software that Upptec supplies is an in-house developed event sourcing system that has also been released as an open source project called Sandthorn [1]. Event sourcing is an architectural model that defines a system's current state as a sequence of events [2]. The storage mechanism used to store all events is called event store. One part of the Sandthorn project is the database driver. The driver is specific to the used database, which means that Sandthorn can be used with any database, given that a driver exists. Today, the project includes a SQL database driver. In most event sourcing systems the event store is a relational database that guarantees high transaction reliability when storing events. Relational databases have several drawbacks when it comes to performance, scalability, concurrency, fault-tolerance and availability [3, 4]. To overcome these problems, NoSQL databases have been developed.

In the near future, Upptec predicts greater data volumes and higher demand for availability. This implies higher demands on the system and the used database.

1.3 Aim

The aim of this thesis is to make a study of NoSQL databases and to investigate if they can be of any use in an event sourcing system. In order to do that it's important to understand how event sourcing works and what the requirements are on a database for being used as an event store. We will also see how NoSQL databases differ from relational databases, what different kind of NoSQL databases there exist today and if they can be of any use in an event sourcing system. The NoSQL database that seems to be most suitable will be used to implement a new event store that can be used in Sandthorn. We will then try to answer the following questions for the event store implementation:

- Querying: Can it support all the needed operations in an easy way?
- Scaling: Is it easy to scale the chosen database?
- Response time: how long will it take to save and retrieve events?
- Is the chosen NoSQL database suitable to use as an event store and can it replace the SQL event store?

1.4 Approach

In the first part of this thesis, a literature study will be made to get a thorough understanding about event sourcing and NoSQL databases. First we will see how an event sourcing system works and what is needed to be stored in such system. Important concepts about NoSQL are presented and as we will see, NoSQL databases can be divided into different categories. With this knowledge, requirements on an event store will be defined and each NoSQL category will be evaluated against the requirements. The most appropriate NoSQL database will then be chosen to implement an event store. The development methodology that will be used to implement the event store is test-driven development. Test-driven development is a process of short development cycles. First a failing test is written, then a minimal amount of code is produced to pass the test and at last the code is refactored to improve the design. When the implementation is done, the event store will be tested together with the Sandthorn project.

At last, performance tests are made on the implemented event store to measure the time is takes to save and retrieve events. The same performance tests will be made on the existing SQL event store and then the two solutions are compared.

1.5 Thesis outline

The thesis is organized as follows:

- Chapter 2 introduces the concept of event sourcing.
- Chapter 3 gives a theoretical background and presents NoSQL databases.
- Chapter 4 gives an analysis of which database that should be further investigated.
- Chapter 5 presents the graph database Neo4j.
- Chapter 6 describes the implementation of an event store using Neo4j.
- Chapter 7 presents the results of the performance tests that have been carried out in order to evaluate the implemented solution.
- Chapter 8 summarizes the thesis.

Chapter 2

Event sourcing

In order to figure out what is needed to be stored in an event sourcing system and how it should be stored, it is important to have a good knowledge of how such a system works.

2.1 Description

The traditional approach to work with data in an application is to use the CRUD (create, read, update, delete) model. An application can create new data, read data from some storage, update the current state with new values or delete existing data. The event sourcing pattern takes another approach. It's intended to capture all changes to an application state as a sequence of events [2], rather than only store the current state of an application. To understand event sourcing it's important to have a basic definition of an event [5]:

- An event is something that has happened in the past.
- An event is immutable. It's something that has happened in the past and therefore it cannot be changed or undone.
- An event has a single source that publishes the event and one or more recipients may receive events. The source publishes the event by notifying other parts of the system that can be interested in when an event has occurred.
- In event sourcing, an event should describe business intent.

Events discussed in this report are associated with aggregates. The term aggregate is taken from the software development technique Domain-Driven Design and refers to a collection of related objects that are treated as a unit and bound together by a root entity [6]. The root entity is also called aggregate root and its responsibility is to control access to its members. All

references to an aggregate should be restricted to the root. Event sourcing is not directly related to Domain-Driven Design even though they are often used together.

To change the state of an aggregate, a command is applied to the aggregate. A command is basically a request for the aggregate to execute an action that changes the state of the aggregate. The aggregate will then process the command and create events to record the state changes. These events can then be written to a persistent store. All events associated to an aggregate are captured in an append-only event stream [7]. To represent the current state of an aggregate, the event stream is replayed in the same order as which the events happened. The aggregate state can be changed by appending new events to the end of the event stream. The event stream is usually stored in an event store, which is described in more detail in the next section.

In theory, the event streams are all you need to recreate the state of an aggregate. In practice, when the event stream contains a large number of events, loading the current state of an aggregate will be inefficient. Using a technique called snapshots can solve this problem. A snapshot is a representation of an aggregate at a given point in time [8]. To retrieve the current state of an aggregate when using snapshots, you find the most recent snapshot and then replay all events in the event stream that were stored after the snapshot was taken. This means that the amount of events to load from the event store is reduced. It's up to the developer to decide when a snapshot should be taken. It can be taken at regular intervals specified by the number of events or at fixed time intervals.

To illustrate the concept of event sourcing, a simple example of an object-oriented event sourcing implementation is given in Listing 1. In this example there is a class, *Person* that represents an aggregate of a person. The *Person* class includes another class called *AggregateRoot*, which provides a unique aggregate id and functionality to commit and save state changes. *Person* has two methods: *change_name* and *change_age*. These methods are the aggregate's commands and are used to change the state of the aggregate. By invoking one of these methods, the aggregate's state changes and an event is applied. When an event is applied, the method *commit* (provided by *AggregateRoot*) is called and the changes of the aggregate are extracted and stored locally. At last, when one or more commits have been applied to the aggregate it should be saved. The *save* method appends all committed events to the event store.

Line 27-30 in the example shows how a *Person* aggregate is being created, two state changing commands applied to it and then saved to the event store. Figure 2.1 shows the resulting event stream of the aggregate.

```
1  class Person
2    include AggregateRoot
3
4    def initialize(name = nil)
5      @name = name
6    end
7
8    def change_name(new_name)
9      @name = new_name
10     name_was_changed
11   end
12
13   def change_age(new_age)
14     @age = new_age
15     age_was_changed
16   end
17
18   def name_was_changed
19     commit
20   end
21
22   def age_was_changed
23     commit
24   end
25 end
26
27 person = Person.new("Kalle")
28 person.change_name("Pelle")
29 person.change_age(26)
30 person.save
```

Listing 1: Event sourcing example

Figure 2.1: Event stream

2.1.1 Event Store

The event store is as the name implies, storage for events. It's free to choose which storage mechanism that should be used. It can be a relational database, a NoSQL database, a file, etc. The structure of an event store is pretty simple. In a relational database this can be represented using a single table with three columns holding values for aggregate id, version number and event data. To optimize the retrieval of the current aggregate version, two tables are used in most cases, one for the events and one for the aggregates. The table representing the aggregates has three columns:

- **aggregate id:** a unique identifier
- **type:** the type of aggregate. In an object-oriented event sourcing implementation, this is the aggregate's class name.
- **version:** current version of the aggregate. Incremented every time an event is applied to the aggregate and indicates how many events have been applied to it since it was first created.

The table representing the events has three columns:

- **aggregate id:** a foreign key pointing to the aggregate table
- **version:** incremented version number
- **event data:** the actual event

Tables 2.2 and 2.1 show an example of what the two tables could look like.

aggregate id	type	version
1	”Person”	3

Table 2.1: Aggregate table

aggregate id	version	event data
1	1	person_created; name = ”pelle”
1	2	name_was_changed; new_name = ”kalle”
1	3	age_was_changed; new_age = 26

Table 2.2: Event table

Event stores are simpler than most storage mechanisms and at its simplest level it only supports two operations. The first operation is to retrieve all events for an aggregate. This operation uses the version number to return the events ordered, which is important when replaying the events to build up the current state. The second operation is to write a set of events to the event store. It is done in the following steps:

1. Check if an aggregate with the unique aggregate id exists. If it doesn't, insert it into the aggregate table and set current version to 0.
2. The version number of the first event to be stored should be equal to the version number in the aggregate table plus one. If it's not, there is a conflict and an exception should be raised. This approach is called optimistic concurrency control.

3. Loop through the set of events and insert them into the event store, incrementing the version number for each event.

4. Update the version in the aggregate table to the version of the last inserted event.

In some event sourcing implementations there is also a need to retrieve all events for the purpose of creating read models. Also here it is important that events are ordered in the same way, as they were stored. In a relational database, a column with an incrementing sequence number can be used to order the events.

When snapshots are used the event store should be able to store a serialized state of the aggregate together with the version number of the aggregate. In a relational database the snapshot table should have three columns, aggregate id, version number and serialized data. The version number represents which version of the aggregate the snapshot represents. The serialized data contains a serialized aggregate at a given point in time. How the aggregate is serialized depends on the system. Table 2.3 shows an example of what a snapshot table could look like. To show what is stored in the serialized data column, data is stored as plain text.

aggregate id	version	serialized data
1	3	name = ”kalle”; age = ”26”

Table 2.3: Snapshot table

Chapter 3

NoSQL

3.1 Theoretical concepts

In this section some theorems and concepts are presented. They are important to understand when comparing different databases and also to see why the NoSQL movement has emerged.

3.1.1 The CAP Theorem

Eric Brewer presented the CAP theorem, also known as Brewer's conjecture, in 2000 [9] and it describes important properties of a distributed system. A distributed system is a collection of independent computers (also called nodes) connected by a network and that appears to the user as a single system. When a database distributes data among the nodes in a distributed system, it's called a distributed database. The theory about how a distributed system works is important to understand because NoSQL databases are typically designed to be distributed. The theorem states that a distributed system cannot simultaneously guarantee consistency (C), availability (A) and partition tolerance (P):

- Consistency means that after an update operation is done, all readers should view the same data. In a distributed database, a consistent write is only done if the data has been updated on all nodes. Then all nodes should contain the same data [10].
- Availability means that all requests to a distributed system received by a non-failing node should result in a response [10].
- Partition tolerance is the ability to still work when a network partition occurs. A network partition is when a network is split into smaller networks, called subnets and any two nodes in the subnets are unable to communicate with each other [10].

At most, two of the three properties can be guaranteed. This means there are three options when designing a distributed system.

1. Consistent and available
2. Consistent and partition tolerant
3. Available and parition tolerant.

The first option means that the database is not distributed so therefore the actual choice is between availability and consistency. When consistency is chosen, the system may not always be available and when availability is chosen, data may be inconsistent sometimes.

3.1.2 ACID properties

ACID refers to a set of properties that describes a reliable database transaction. A database that guarantees all these properties is considered to be reliable. The ACID acronym stands for [11]:

- **Atomicity** In a transaction all or none operations will complete. If one operation in the transactions fails, the entire transaction fails.
- **Consistency** A transaction will bring the database from one consistent state to another.
- **Isolation** Operations in one transaction cannot access data that is currently modified by another transaction.
- **Durability** Once a transaction has been completed, it will not be lost.

3.1.3 BASE properties

Traditional relational databases focus on the ACID properties and choose consistency over availability for partitioned databases. An alternative to ACID is BASE [12], which stands for:

- **Basically available:** The database works basically all the time, despite partial failures.
- **Soft state:** The state of the database may change without any input.
- **Eventually consistent:** The database may temporarily be inconsistent but will eventually be consistent.

Brewer introduced the BASE properties to achieve availability instead of consistency. While ACID is pessimistic and forces consistency at the end of each transaction, BASE is optimistic and accepts that data can be inconsistent at some time but will eventually be consistent. Both performance and scalability can be improved when following the BASE properties [12].

3.1.4 Distribution models

NoSQL databases are often designed to be horizontally scalable[1]. To achieve this, data need to be distributed among several nodes in a cluster. There are two ways to distribute data: replication and sharding [13]. Replication is the process of copying the same data over multiple nodes, so all data can be found in multiple places. Sharding is the act of dividing data into separate nodes.

Sharding

Sharding is the process of distributing data among shards, where a shard is a server or a set of servers, containing a part of the entire data. Data is distributed according to a key field and each shard is responsible for data for specific intervals of the key. When sharding is used the amount of storage needed is reduced because instead of storing duplicate data in all nodes, one or multiple nodes are responsible for storing a shard. By distributing data according to this method both read and write performance can be improved [13]. Many NoSQL databases offer auto-sharding, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard.

Master-Slave Replication

Master-slave replication is a model where data is replicated across multiple nodes. One node in the cluster is the master and is usually responsible for processing any updates of data [13]. The other nodes act as slaves and a replication process synchronizes them with the master. A master node can be chosen manually or automatically. When using automatic appointment the cluster can automatically appoint a new master if a current master fails. This model is most helpful for scaling when you have a read-intensive dataset. By adding more slaves and use a configuration that routes all reads to the slaves, it's easy to scale horizontally. By replicating data to several nodes higher availability is achieved because if one node goes down, the same data can be found in another node. The disadvantage with this replication model is that the amount of storage needed is increased with how many nodes the data should be replicated to. Another weakness is that the master is a single point of failure for writes.

Peer-to-Peer Replication

By using master-slave replication, read scalability can easily be improved but write scalability is limited to the master. The master is a bottleneck and a single point of failure. Peer-to-peer replication solves these problems by not having a master. All nodes are equal and can accept writes. The nodes

[1] See 3.2 Scalability

coordinate to synchronize their copies of data. When writes are allowed to multiple nodes, there is a risk that two people try to update the same data at the same time, a write-write conflict. These conflicts need to be solved in order to keep data consistent. Peer-to-peer replication tends to favor availability over consistency, violating ACID properties.

3.2 Relational databases

Edgar Codd invented the relational database model in 1970 while he was working for IBM [14]. In a relational database, data is organized into tables, also called relations. A table consists of rows and columns where a row represents a unique instance of data and a column represents an attribute to that instance. Each row can be identified by a primary key. Together with this model IBM introduced SQL (structured query language) to manage data in relational databases. Software systems that use relational databases are known as relational database management systems (RDBMS).

Even though relational databases have been the predominant choice since it was introduced it has several limitations [4].

Scalability

Scalability is the ability of a system to grow the capacity when the workload increases. Relational databases do not support high scalability. A relational database can be scaled vertically, which means upgrading the hardware. At some point the database must be distributed across multiple machines and relational databases are not designed to run on clusters. The partitioning of data causes problems because joining tables in a distributed system is not easy and also decreases the performance. The other way to scale a database is horizontally, which means adding more nodes to a system. Most of the NoSQL databases are designed for this type of scaling.

Data model

In a relational database all data needs to be converted into tables and rows. If the data cannot easily be converted, the structure of the database can be complex and difficult to work with. The problem is called impedance mismatch and has been a problem for developers during a long time. In the 1990s many object-oriented databases were developed to store in-memory data directly to disk but they never became really popular.

Query language

Relational databases use SQL for managing data. This works well when data is structured and organized into fixed table. To use SQL with unstructured data can be highly complex and can result in a large amount of code.

Large feature set

Relational databases provide a large set of features (multi platform support, support for transactions, development and administration tools, etc.) and data integrity[2]. In most applications, all of the features are not used and therefore only add cost and complexity. Proponents of NoSQL databases also say that relational databases focus on data integrity instead of the data it self.

3.3 NoSQL

Carlo Strozzi who led the development of a database called "Strozzi NoSQL" in 1998 was first to use the term "NoSQL" [13]. The name was used to indicate that the database didn't used SQL as query language. Eric Evans reintroduced the term in 2009 to name an event for open-source, distributed databases [15] and since then the term has been widely used. Although there is no formal definition of NoSQL most people say that it means "Not Only SQL".

The rise of NoSQL databases has been driven by the need to handle large volume of data that came with new web technologies. Web application companies like Amazon and Google developed DynamoDB and BigTable, respectively, to handle this problem, which have had a big influence on today's NoSQL databases [12].

As stated before, there is no formal definition of NoSQL but most of them have the following characteristics [13]:

- Not using the relation model or the SQL language
- Designed to be distributed
- Open-source
- Designed for the 21st century web, which requires a large amount of data to be stored
- Schema-less
- Horizontally scalable

3.4 NoSQL categories

It's not easy to classify NoSQL databases into categories because they are designed to solve different kinds of problems. One common approach is to divide them into four categories based on their data model [13].

[2]refers to the overall completeness, accuracy and consistency of data

3.4.1 Key-Value Store

A key-value store has the simplest data model among the NoSQL categories and is based on the abstract data type associative array. Data is represented as a collection of key-value pairs. The key appears just once in the collection and identifies the stored data. The value is the data itself and can be structured or unstructured. Many key-value stores have great performance because it's very efficient to query data by keys. It is also easy to scale this type of database because the data can easily be partitioned on different machines when the data grows. The disadvantage is the possibility to query data if the key is not known. Amazon's DynamoDB is a popular key-value store that uses concepts from their Dynamo paper [16], which describes how to implement a highly available key-value store. The Dynamo paper has inspired many other key-value stores, such as Riak, Cassandra and Voldemort.

DynamoDB

DynamoDB has a data model containing tables, items and attributes. A database is a collection of tables. A table is a collection of items and each item is a collection of attributes. To manage data, four operations are provided, get, put, delete and update. All these operations are limited to one key-value item at a time.

Table 3.1 shows a *Users* table with the attribute *Email* as its primary key. The primary key uniquely identifies an item and is the only required attribute. Note that except the email attribute, the items don't need to have the same attributes.

Users
Email = kalle@gmail.com, Name = "Pelle", Age = "25"
Email = pelle@gmail.com, Name = "Kalle", City = "Stockholm"

Table 3.1: DynamoDB example

DynamoDB is designed to be highly available and scalable. This is achieved by portion and replicate data using consistent hashing. The database is also designed to be eventually consistent which means that an update operation returns before the update has propagated to all replica nodes. This can result in nodes having different versions of data and to solve these conflicts the concept of vector clocks[3] is used.

[3]algorithm used in a distributed system to determine a partial ordering of events

3.4.2 Document Database

In a document database, data is stored in the form of documents. A document encapsulates key-value pairs in some semi-structured format like JSON, XML, YAML or BSON. The documents don't have a uniform structure and the types of the values can be different for each document. A document contains a unique field ID and the values can be a variety of types, including other documents. Examples of document databases are MongoDB, CouchDB and Couchbase.

MongoDB

MongoDB is the most widely used document database [17]. It stores data in form of BSON[4] documents and each document belongs to a collection. The documents in a collection are related to each other but can vary in structure because MongoDB is schema-free. A MongoDB server can have multiple databases where each database contains one or more collections.
Figure 3.1 shows an example of a MongoDB document. The document belongs to a collection named *Users* and contains fields and values. The values in a document can be any of the supported BSON types, including other documents as can be seen in the value of the *address* field. The field *_id* is required in all documents and is used as a primary key. Its value is immutable and uniquely identifies a document in a collection.

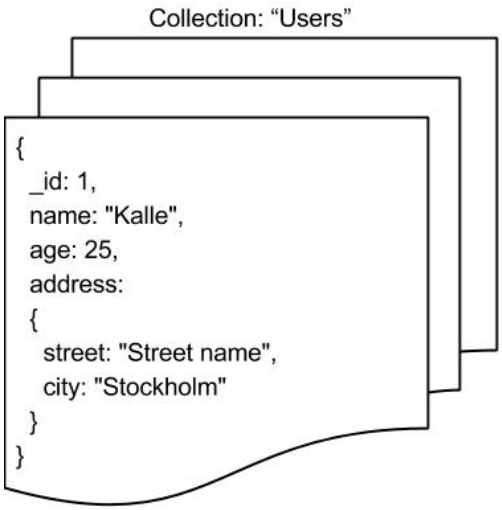

Figure 3.1: MongoDB example

To support scalability, MongoDB uses both replication and sharding. It uses a variant of master-slave replication called replica sets. A replica set is a group of servers containing the same data. One server is the master that receives all write-operations. The data is then replicated to the other servers, the slaves. When the master is not available, a new master can

[4]Binary-encoded serialization of JSON-like documents

be elected automatically among the slaves. To scale for reads, more slaves can be added to a replica set and then direct all reads to the slaves. To scale for writes, it doesn't help to add more nodes and therefore sharding is used. MongoDB is a consistent database by default but can be configured by specifying how many slaves a write operation should be replicated to before it returns. MongoDB has support for ACID transactions at the level of a single document. A transaction that modifies multiple documents is not possible.

3.4.3 Column-Oriented Store

In a column-oriented database, data is stored in columns. The approach to store data by column instead of row comes from analytics and business intelligence where there is a need to perform parallel processing on very large dataset across different machines. In a column-oriented database, data can be stored effectively. Instead of storing null values if a column doesn't exist, the column is simply not stored. The main inspiration of many column-oriented is Bigtable, developed by Google. Other column-oriented databases are Apache Cassandra, HBase and Accumulo.

Apache Cassandra

Apache Cassandra was developed and released by Facebook in 2008. The database adopts concepts of both Amazon's DynamoDB and Google's Bigtable. Cassandra is a distributed database built to handle very large amounts of structured data and be highly available. Cassandra structures data into:

- Columns: A key-value pair with a timestamp attached to it.
- Rows: A collection of columns linked to a key. Each row can have a different number of columns.
- Column families: A set of related rows.

Figure 3.2 shows an example of a column family, where all rows are related to some type of user. The first column in the row is the row key, which in this case is an email address. All other columns are key-value pairs, where the keys in this example are *name*, *age* and *city*. A timestamp is also attached to each column but is not shown in the figure. The timestamps are used to expire data, resolve write conflicts etc.

Cassandra replicates data to several nodes in a cluster in order to provide high availability. As the CAP theorem dictates, there is a necessary trade-off between data consistency and availability. Cassandra lets you choose what kind of trade-off you are willing to make. Availability and consistency are tunable by altering three values, N, W and R. N is the number of nodes data should be replicated to, W is the number of nodes that must complete

kalle@gmail.com	**name**	**age**	**city**
	"kalle"	30	"Stockholm"
pelle@gmail.com	**name**	**age**	
	"pelle"	25	
nisse@gmail.com	**name**	**age**	
	"nisse"	27	

Figure 3.2: Cassandra example

a write to be considered successful, R is the number of nodes that must respond successfully to a read.

It's easy to scale a Cassandra cluster because there is no master node. Both write and read capacity can be improved by adding more nodes to a cluster.

Cassandra does not offer ACID transactions with rollback or locking mechanism as in relational database. A write is atomic, isolated and durable at the row-level, which means that updating multiple columns for a given row key is considered as a single write operation and will either succeed or fail.

3.4.4 Graph Database

In a graph database, data is represented as a graph. Data is stored in a graph's nodes and edges. A node represents an entity and an edge represents a relationship between two nodes. Graph databases are suitable to use when data is highly connected. They have their strength in traversing the graph by following relationships and can retrieve connected data quickly. Many graph databases differ from other NoSQL solutions by providing ACID transactions to ensure consistency. An operation in a graph involves connected nodes and therefore can data not be distributed on different servers. Some solutions, such as Neo4j, are still able to scale by using master-slave replication. Examples of the most popular graph databases are Neo4j, OrientDB, FlockDB and Titan.

Chapter 4

Analysis

4.1 Event store requirements

An event store that should be used in the Sandthorn project has to meet the following requirements. The event store should:

1. Be able to store state changes as a sequence of events, chronologically ordered.
2. Be able to read events of individual aggregates, in the order they were persisted.
3. Be able to read all events, in the order they were persisted.
4. Be able to write a set of events in one transaction, either all events are written to the event store or none of them are.
5. Be implemented in Ruby because it should be used in the Sandthorn project.

4.2 Chosen solution

In a key-value store an event can be modeled by constructing the key as a combination of the aggregate id and the version number, which can be seen in figure 4.1. The value can then contain the event data in some serialized form. By using this kind of composed key, the events for an individual aggregate can be stored in a chronological order. To retrieve data from a key-value store the key has to be known. It could be possible to retrieve events of an individual aggregate by just having knowledge about the aggregate id and then combine it with an incrementing version number until a key is not found. The problem is when retrieving events regardless of aggregate because then no part of the key is known and it's not possible to retrieve them in the order they were stored. Many of the key-value stores do not

offer transactions. When there is a need to save multiple events and a failure occurs when saving one of them, the operation cannot be rolled back.

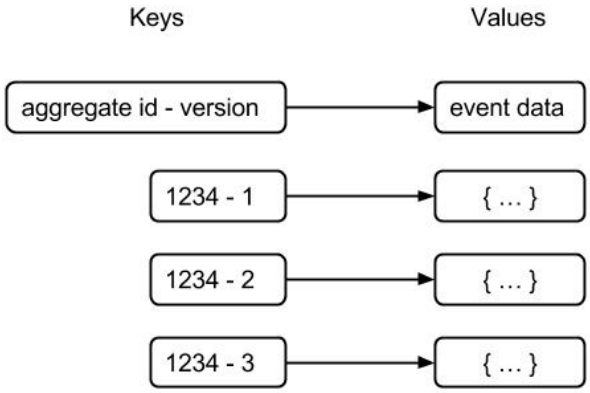

Figure 4.1: Events in a key value store

By using a document database, all events belonging to an aggregate can be stored in a single document or an event can be stored as a single document. Figure 4.2 shows examples of both these ways. A document can have a version field to store the events in a chronological order. When all events belonging to an aggregate are stored in the same document, a query can be made to collect all these ordered by the version number. The events returned by that query will then have the same order, as they were stored. When multiple events are stored in a single document it's not possible to retrieve all events in the right order. One possibility is to retrieve all events and then sort them by a timestamp but that would be very inefficient. Since most document databases support ACID transactions within a document, a set of events can be written in one transaction.

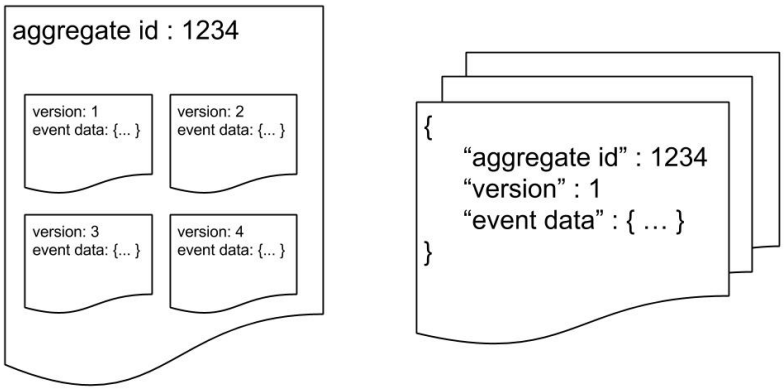

Figure 4.2: Events in a document database

In a column-oriented database, events could be saved as columns and all events belonging to an aggregate could be organized within a row. Figure 4.3 shows an example of how events could be modeled. Each row contains all events belonging to an aggregate. The row key is the aggregate id and

the events are stored in the columns. Each column is a key-value pair where the key is the version number and the value contains the events data. When a new event has to be stored, a new column can just be added because the number of columns can be different for each row. To retrieve events from a single aggregate, the row key has to be known, which is the aggregate id. The columns can then be ordered by the key to collect the events in the right order. A set of events can be written to a row by storing each event in a new column and since many column-oriented databases offer transaction within a row, this can be done in a single transaction. The problem with column-oriented databases is the same as with document databases, there is no easy way to retrieve all events in the order they were stored.

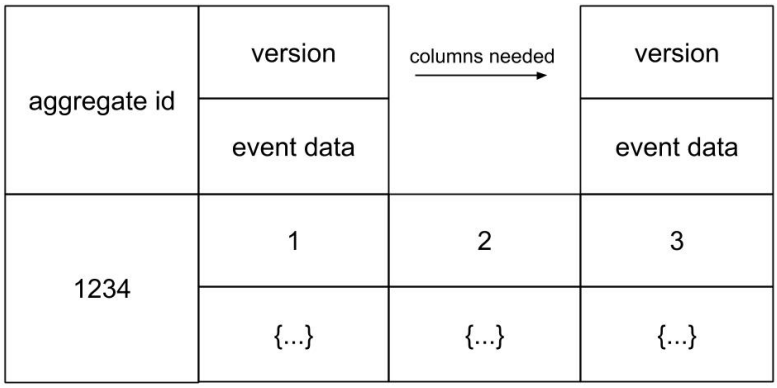

Figure 4.3: Events in a column-oriented database

Figure 4.4 shows how events could be modeled as nodes in a graph database and then connected by edges to store them as a sequence. An aggregate node can be used as start node to retrieve events from a single aggregate by following the edges from one aggregate event to the next. If all events also are stored as sequence, they can be retrieved in the order they were stored. Most graph databases support ACID transactions and therefore a set of events can be written in a single transaction.

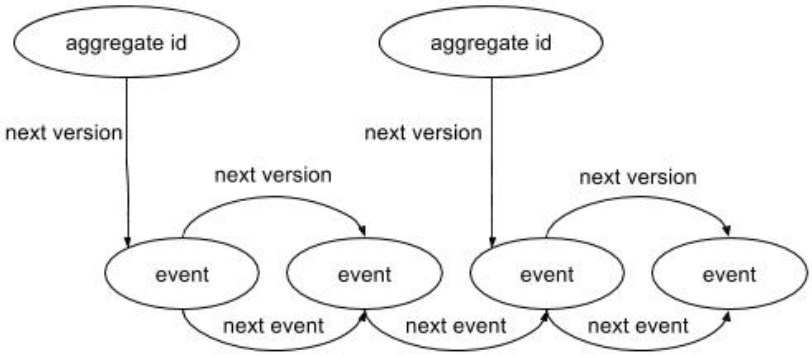

Figure 4.4: Events in a graph database

Requirement	Key-value	Document	Column	Graph
1	Yes	Yes	Yes	Yes
2	Yes	Yes	Yes	Yes
3	No	No	No	Yes
4	No	Yes	Yes	Yes

Table 4.1: Requirement table

Table 4.1 summarizes how databases in each NoSQL category seem to meet the requirements presented in 4.1.The last requirement is not presented in the table because it depends only on database and not on the category. A graph database is the only one that seems to meet all requirements of an event store. Neo4j is the most widely used graph database, mature and is able to scale. It can also be used from Ruby, which was the last requirement. Therefore the choice has been made to make a further study on Neo4j. The following parts of the report will present Neo4j in more detail and an event store will be implemented.

Chapter 5

Neo4j

Neo4j is an open-source graph database developed by Neo Technology and was initially released in 2007. The development of the database started already in 2000, when the founders of Neo4j were having problem with relational databases. Neo Technology describes it as "a robust, scalable and high-performance database, Neo4j is suitable for full enterprise deployment or a subset of the full server can be used in lightweight projects" [18].

Neo4j comes in two editions: community and enterprise. The community edition is free and can only run a single instance of the database. The enterprise edition requires a license but enables features such as clustering, hot backups and advanced monitoring.

5.1 Data model

At the most basic level, a graph database consists of nodes and relationships. In Neo4j, the data model is a property graph. The property graph is made up of nodes, relationships and properties [19]. Both nodes and relationships can have properties. A property is a key-value pair where the key is a string and the value is a primitive value or an array of primitive values. A node with properties can be thought of as a row in a relational database, an entity with one or more attributes. The property of a relationship is often used to quantify the strength of the relationship and can be used when querying the database to find specific patterns in the graph.

Relationships connect nodes and are used to find related data. A relationship has always a direction, a type, a start and an end node. A node can have zero or more labels. Labels are used to group nodes into sets and make it easier and more efficient to query the graph. By using labels in a query, the data set is limited to a sub-graph instead of the entire graph. Labels are also used when indexes and constraints are defined.

Figure 5.1 shows an example of how data can be modeled as a graph. It contains three nodes and two relationships. A node has either a *Person*

or a *Movie* label and a set of properties. The nodes are connected by relationships that have a type of *ACTED_IN* or *DIRECTED*. The *ACTED_IN* relationship also has a property attached to it.

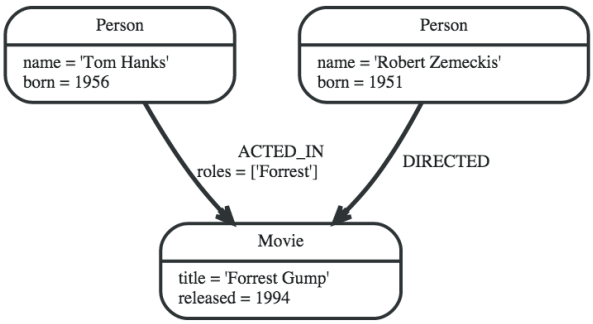

Figure 5.1: Neo4j graph example

Source: neo4j.com

A graph is often queried by traversing it, which means visiting nodes and following relationships according to some specified algorithm. To find a certain node or relationship according to a property, indexes should be used instead of traversing the entire graph.

5.2 Query Language

There are several languages for querying property graphs and as of today, there is no agreed-upon standard language for graph query languages. Neo4j has its own query language called Cypher but also has support for the RDF query language SPARQL and the imperative, path-based query language Gremlin [18].

Cypher is a declarative query language that shares some traits with SQL and uses similar keywords to run operations. It was created to be simple to use for a range of users, such as software developers business analysis and technical architects. As in most query languages, a Cypher query is composed of clauses. A query can start by retrieving a large amount of nodes from the graph and then return a sub-collection of the nodes, also called a sub-graph. The most commonly used clauses in Cypher:

- MATCH is the primary way of retrieving data from the database and is used to search for patterns in the graph.
- WHERE filters the properties on nodes and relationships found in matched pattern.
- RETURN specifies which nodes, relationships or properties of the nodes or relationships that should be returned to the client.
- CREATE create nodes and relationships with properties.

- DELETE: removes nodes, relationships and properties.

The following query will match two nodes, filtered by a label and a node property. When the nodes are found, a relationship is created between them and then the relationship is returned.

```
MATCH (a:Person),(b:Person)
WHERE a.name = 'A' AND b.name = 'B'
CREATE (a)-[r:RELTYPE]->(b)
RETURN r
```

5.3 Transactions

Neo4j is an ACID-compliant database. Once a transaction starts, every following operation will succeed or fail as an atomic unit. All or no operations will complete. In Neo4j, all database operations that modify the graph or indexes must be run in a transaction. When a database operation begins, it will run within an existing transaction or Cypher will create a new one. Read operations can be done without being wrapped in transactions. The execution of a transaction is done in the following steps [18]:

1. Start a transaction.
2. Perform operations.
3. Mark the transaction as successful or failed.
4. Finish the transaction.

The transaction will not release locks or memory until it has been finished and therefore the last step is very important. By finishing the transaction, a commit or rollback will be executed depending on the success status that was set in the previous step.

5.4 Indexes

Neo4j is a schema-optional database. It can be used without any schema but is recommended to use in order to increase performance. Performance is increased by creating indexes over properties for a given label. These indexes make it efficient to look up nodes in the database given a certain property. Indexes are automatically managed and updated when the graph is updated. When an index is created, it will be populated in the background and is not immediately available for querying. In Neo4j 2.0, unique constraints were introduced. By using unique constraints based on labels, it's ensured that a node's properties are unique and any attempt to break a constraint will be denied.

5.5 Availability

Neo4j provides different strategies to support high availability and to avoid a single point of failure [18].

Online backup

A single database instance is used and all data is replicated into a local copy of the database. In the event of failure, the backup can be mounted into a new database instance and integrated into the application.

Online backup high availability

One master instance and one backup instance are running. The backup instance is listening to online transfers of changes from the master instance. In the case of failure, the backup instance is already running and can directly become a new master instance.

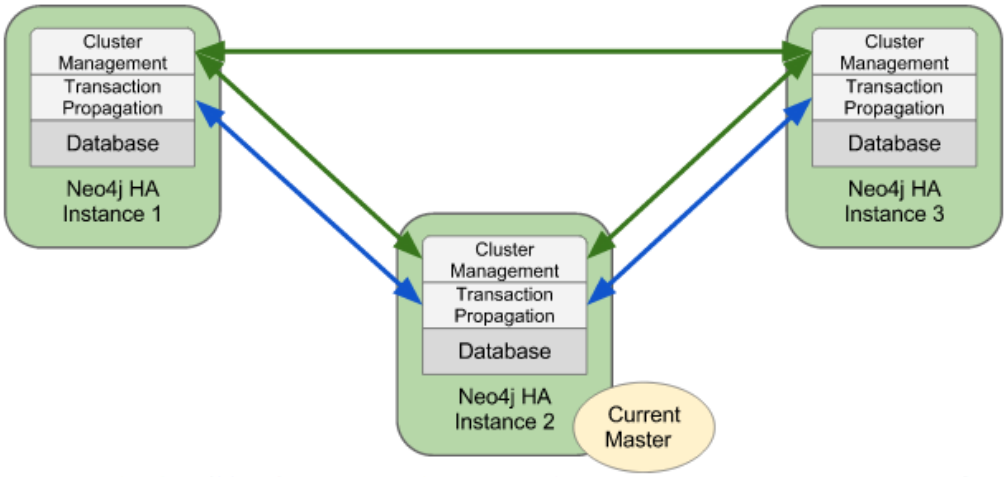

Figure 5.2: Neo4j High availability cluster

Source: neo4j.com

High availability cluster

The Neo4j high availability (HA) cluster solution uses a master-slave architecture [20] as is shown in figure 5.2. In contrast to the traditional master-slave replication setups Neo4j HA can handle write requests on all slaves. Each instance in the cluster holds the entire dataset of the database and can respond to any query request. When a slave handles a write request, it will first synchronize the write with the master in order to preserve consistency. The write is first committed at the master and then at the slaves. A write will then be pushed out to the rest of the slaves in the cluster. All updates will propagate from the master to the slaves eventually and therefore a write

may not be immediately visible on all slaves. In case of a conflict in the cluster it is always the master instance that decides how the conflict should be solved. To achieve high availability, the cluster elects a new master if the current one goes down.

Chapter 6

Neo4j event store

6.1 Data model

To implement an event store that uses Neo4j we will first have to define how the data should be modeled. This will be done in a few steps. We first define a minimal model that can be used in the Sandthorn project and then we extend it to make it easier to work with and improve the performance. The data that is needed to be stored in the event store are events. For each event, this is what is needed to be stored:

- Aggregate id: unique aggregate identifier
- Aggregate version: incremented version number
- Event id: unique event identifier
- Event name: name of the event
- Event data: the actual event
- Timestamp: time when the event occurred
- Aggregate type: type of aggregate the event belongs to

This event data can be stored in a single node, where all fields are stored as properties. All event nodes need to be connected by a relationship in order to keep them in a chronological order and therefore a relationship called *next event* is introduced. This minimal model can be seen in figure 6.1.

The problem with this model is that there is no easy way to retrieve events from a single aggregate. It can be done by traversing the graph and then filter the nodes by the aggregate id but that's not efficient. To make this easier a new type of node is added, an aggregate node. In this node, aggregate id, aggregate version and aggregate type are stored as properties. The aggregate version in this case indicates how many events have been

Figure 6.1: Minimal event store model

applied to an aggregate since it was first created. When an event is applied to an aggregate, the version number is incremented by one.

Two new relationships are also added, called *next aggregate event* and last aggregate event. An aggregate node has both of these as outgoing relationships. The *next aggregate event* relationship points to the first event in the stream belonging to that aggregate and then the same type of relationship connects the rest of the events. The aggregate node can be used as a start node when events belonging to an aggregate should be retrieved. By following the *next aggregate event* relationships, aggregate events can be retrieved in the order they were stored. The *last aggregate event* relationship is added to quickly find the last inserted event, which is needed every time a new event has to be stored.

The next node to be added is a node of type event root. The event root is a single node for identifying the start of the event stream. It has two outgoing relationships: *next event* and *last event*. The *next event* relationship points to the first event in the stream and *last event* relationship points to the last inserted event. The *last event* relationship is used to quickly find the last inserted event instead of traversing the whole graph.

In the Sandthorn project there is also a need to retrieve aggregate events that are of a certain aggregate type. Therefore an aggregate type node is added to the model, which only holds an aggregate type as a property. A relationship called *is of type* connects an aggregate node to an aggregate type node.

Figure 6.2 shows how the event store data is chosen to be stored in an event store using Neo4j. To summarize it, there are four different types of nodes: event root, aggregate, aggregate type and event. These nodes are connected by some of the following relationships: *next event*, *last event*, *next aggregate event*, *last aggregate event* or *is of type*.

This data model will not support the snapshot technique described in section 2.1. Snapshots are only used to improve performance and therefore not a requirement for an event store implementation.

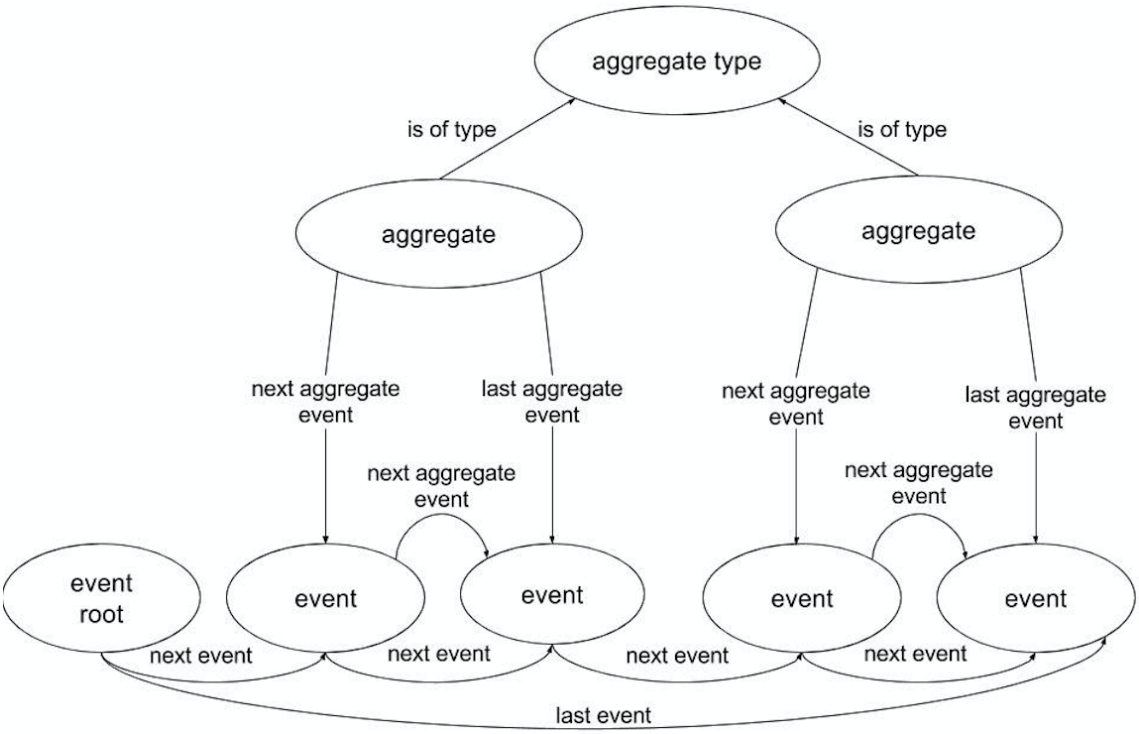

Figure 6.2: Event store data model

Indexes

To improve performance when reading from the graph, indexes are used on the aggregate id property for aggregate nodes, event name and event id for event nodes and on aggregate type for the aggregate type nodes. The following Cypher queries are used to create the indexes over a property with a certain label:

```
CREATE INDEX ON :Aggregate(aggregate_id)
CREATE INDEX ON :Event(event_name)
CREATE INDEX ON :Event(event_id)
CREATE INDEX ON :AggregateType(aggregate_type)
```

Indexes are only created once, when the event store starts up for the first time. Then they are automatically changed and updated when the graph is changed. When making a query there is no need to specify which index that should be used, Neo4j will figure that out by itself.

6.2 Implementation

The Sandthorn project is written in Ruby and therefore the event store is implemented in Ruby. To access Neo4j from a Ruby application there are a few options. It can either be accessed directly by using the REST API or a language driver can be used. Neo4j has drivers for all major programming

languages including Ruby. The language drivers use the REST API under the hood as well but make it easier to use. The most popular language drivers for Ruby are called Neography, Neo4j-core and Neo4j.rb.

- Neography is only a thin wrapper for the REST API.
- Neo4j-core provides basic communication with the database and has also support for classes and methods related to nodes and relations.
- Neo4j.rb uses Neo4j-core to communicate with the database but also provides extra modules that can be used in web frameworks.

The chosen one to use in this implementation is the Neo4j-core driver. In the event store there is no need for extra modules and support for classes and methods makes the implementation easier.

The database can be run in embedded mode or server mode. In the embedded mode the application accesses the database using the Java API. This mode requires using JRuby[1] but because the rest of the application doesn't use JRuby, the embedded mode is not a choice. Therefore the server mode is used, which is also the recommended choice.

During the implementation, a test framework has been used to test all the supported operations and control that the event store behaves as it should.

6.3 Operations

In this section the most important supported operations are described in the Cypher language. All requirements in section 4.1 will be fulfilled with these operations. The parameters are arguments used in an operation.

Save events

The operation used to store a set of events belonging to an aggregate. Parameters:

- Events: a set of events, each event contains aggregate version, event name, event data and timestamp
- Aggregate id: unique aggregate identifier
- Originating version number: version of the aggregate when the events were generated.
- Aggregate type: the type of aggregate

If the originating version number is equal to zero it means that the aggregate does not exists and needs to be created:

[1] implementation of the Ruby programming language atop the Java Virtual Machine

```
CREATE (n:Aggregate { aggregate_id : 'aggregate_id',
  aggregate_version : 0})
```

If the aggregate type does not exists, create it:

```
CREATE (n:AggregateType { aggregate_type: 'aggregate_type'})
```

Then create a relationship between the aggregate and the aggregate type:

```
MATCH (a:Aggregate),(b:AggregateType)
WHERE a.aggregate_id = 'aggregate_id'
AND b.aggregate_type = 'aggregate_type'
CREATE (a)-[r:IS_OF_TYPE]->(b)
```

Insert the events by performing the following operation for each event:

```
MATCH (a:Aggregate {aggregate_id: 'aggregate_id'})-
  [last_a_rel:last_aggregate_event]->(last_a),
  (root:event_root)-[last_e_rel:last_event]->(last_e)
CREATE (last_a)-[:next_aggregate_event]->(e:event{to_insert})
  <-[:last_aggregate_event]-(a),
  (last_e)-[:next_event]->(e)<-[:last_event]-(root)
DELETE last_a_rel, last_e_rel
```

At last, the aggregate version number is updated to the version number of the last inserted event:

```
MATCH (a:Aggregate { aggregate_id: 'aggregate_id' })
SET a.version_number = 'current_version_number'
```

Get aggregate events

Operation used to retrieve events belonging to an individual aggregate. Parameters: aggregate id

```
MATCH (a:aggregate)
WHERE a.aggregate_id = 'aggregate_id'
WITH (a)
MATCH (a)-[:next_aggregate_event*]->(e: event)
RETURN e
```

In the second match clause there is an asterisk (*) symbol given after the relationship type. This means that the query will match event nodes that are several relationships away from the found aggregate node.

Get events

Operation used to retrieve events regardless of aggregate id. This operation takes a lot of parameters but all of them are optional. The query is built up in different ways depending on which parameters that are specified. The query described here is an example when all parameters are specified.

- Aggregate types: array of aggregate types
- Take: Number of events that should be returned
- After event id: retrieve events after a specified event id, default is to start from the event root.
- Include events: array of event names
- Exclude events: array of event names

```
MATCH (start:event {event_id: 'after_event_id'})
WITH (start)
MATCH (start)-[next_event*]->(e:event),
  (t:aggregate_type)<-[:IS_OF_TYPE]-(a:aggregate)
  -[:next_aggregate_event*]->(e:event)
WHERE t.aggregate_type IN 'aggregate_types'
AND e.event_name IN 'include_events'
AND NOT e.event_name in 'exclude_events'
LIMIT 'take'
RETURN e
```

Get all aggregate types

Operation used to retrieve all types of aggregates.

```
MATCH (a:aggregate)
RETURN DISTINCT a.aggregate_type
ORDER BY a.aggregate_type
```

Chapter 7

Evaluation

An event store using Neo4j has been implemented and can now be used with the Sandthorn project. In this chapter the implemented solution will be evaluated. The focus will be on measuring the response time of some real scenario operations. All tests cases will be performed on both the event store using Neo4j and the SQL event store using PostgreSQL. The purpose of this is to determine if the implemented solution can be a suitable alternative to the existing solution. It should be noted that the Ruby application communicates with Neo4j via the REST API and with PostgreSQL via binary protocols. The comparison is not really fair but this is how the databases will be used in production and therefore it doesn't matter how the application communicates with the database.

Test environment

The performance tests were run on a machine with the following specifications:

- Processor: 2,4 GHz Intel Core i5
- Ram: 8 Gb 1600 MHz DDR3
- Disks: 1 x 250 Gb SSD

Performance test

A testing framework has been developed in order to run the different test cases. This testing framework can include any event store that can be used with Sandthorn. It generates aggregates and events that can be stored and retrieved in the different test cases. In this section four different test cases are presented. These test cases reflect the most commonly used operations. In all cases the tests are run ten times and the mean value is calculated and presented as the result.

The first test measures the time taken to write a set of events to the event store and the number of events are increasing. To write to the event store, the "save events" operation is used. The operation saves a new aggregate with a different number of events and therefore includes a creation of an aggregate.

Figure 7.1 shows the average response time for writing a set of events to the event stores. As we can see, the SQL event store has best performance and the difference is increasing as the number of events is increasing.

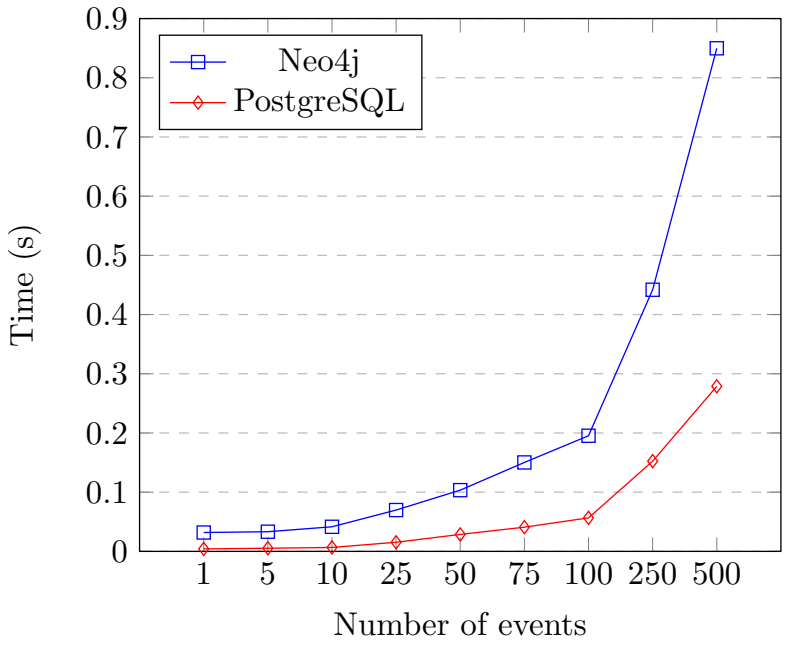

Figure 7.1: Insertion time

The second test measures the time taken to retrieve all events for an individual aggregate. The event store is first populated with aggregates that have a different number of events. To retrieve an aggregate's event, the "get aggregate events" operation is used and an aggregate id passed as parameter. The results of the second test case can be seen in figure 7.2. The event stores have similar response time but when the number of events is increasing, the SQL event store has best performance.

The third test measures the time taken to retrieve a sequence of events independently of aggregate. In this test, the "get events" operation is used and the only specified parameter is the take parameter, which limits the number of event to retrieve. Figure 7.3 shows the result of the third test case. This result shows that the two solutions have similar performance as well but the SQL event store is slightly better.

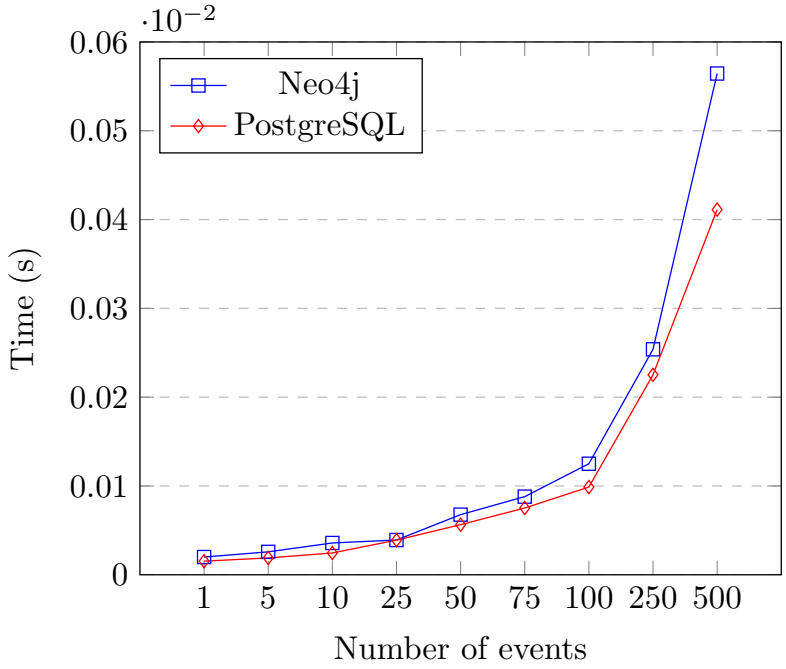

Figure 7.2: Response time when retrieving events from a single aggregate

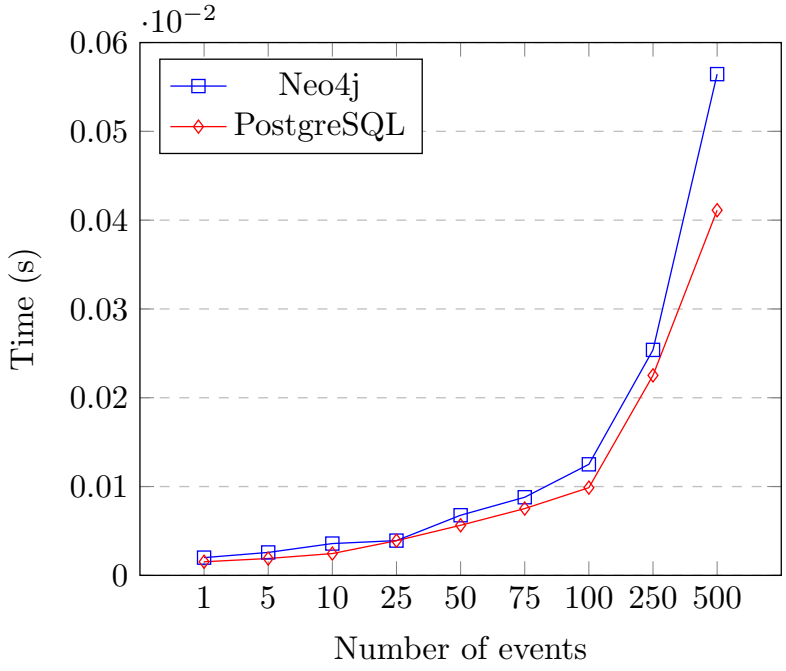

Figure 7.3: Response time when retrieving events

The fourth test measures the time taken to retrieve a sequence of events and only includes events of a single aggregate type. This is done by using the "get events" operation and specifying aggregate type and take as parameters. The aggregate type parameter contains a single aggregate type

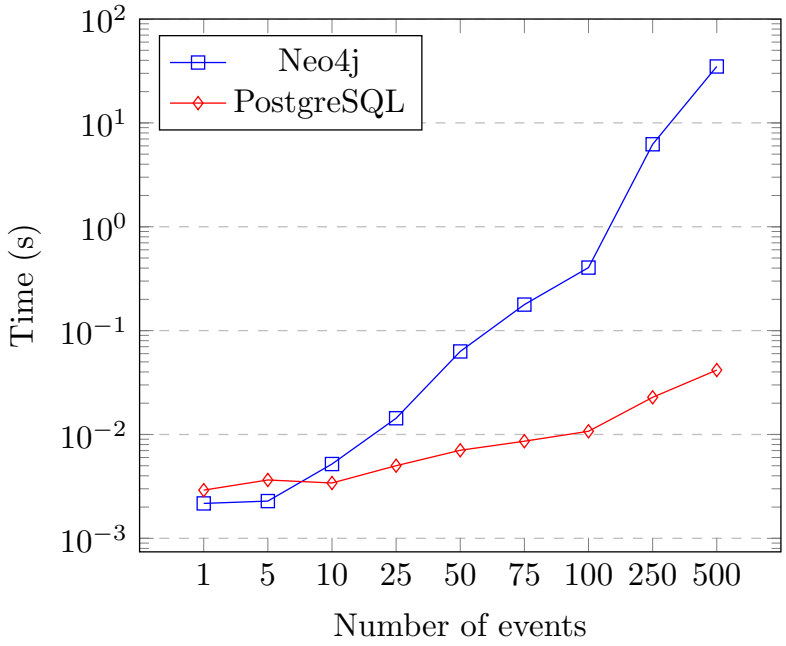

Figure 7.4: Response time when retrieving events that belongs to an aggregate type

and then the take parameter is varied.

Figure 7.4 shows the result of the fourth test case. When the number of events is small, the two solutions have quite the same performance even though the one using Neo4j is slightly better. When the number of events is increased, the difference becomes very big and the one using Neo4j doesn't perform well at all.

Conclusion performance test

In all test cases, the event store using PostgreSQL has better performance than the one using Neo4j. The overall expectations were that the Neo4j event store should be slower than PostgreSQL when saving events but faster when retrieving events. The first test case showed us that there was a small difference between the two solutions when the number of events was small. The set of events that are stored in an event sourcing system is often quite small and therefore the performance of saving a large set of events is not that important. This test result was expected because when saving a set of events there are many nodes and relationships that are needed to be stored. In the fourth test case, the event store using Neo4j was very slow when it should retrieve a higher number of events. This is probably because it has to traverse many nodes to find events of a certain type. Rather than just follow a path and retrieve all nodes in that path, it has to filter by the type property. In the second and third test case, only a single path has to be found and then all nodes in it can be returned. The results of these test

cases were not expected. The expectations were that it should be very quick to retrieve a single path of nodes and that it should have better performance than the PostgreSQL event store.

The snapshot technique that was presented in section 2.1 could have been implemented to improve the performance. However, the same technique can be used in the event store using PostgreSQL and therefore are tests without using snapshots the ones that matter.

Chapter 8

Conclusion and future work

In this report we have seen how an event sourcing system works and figured out what requirements we should have on a database for being used in such a system. NoSQL databases were introduced and we have seen what features they have and how they differ from relational databases. The different NoSQL categories have been presented and we figured out that events could be modeled in all categories but the lack of transactions and the need to store events in a certain order made many of them unsuitable for use as an event store. The NoSQL solution that seemed to fit well was a graph database and Neo4j was explored in more detail. The Neo4j database can be run in a cluster mode and therefore both performance and availability can be improved.

An event store using Neo4j was implemented and evaluated. It was easy to model the event store data in a graph but there were some queries that became a bit complex. When the implemented event store was evaluated it had poorer performance than the existing one using PostgreSQL.

We should not forget that Neo4j is a relatively new database and new versions are released constantly. Therefore it will be interesting to run the performance test again at a later point in time to see if it has improved.

New NoSQL databases pop up all the time and the domain is evolving fast. In the future, there will maybe be a database in some other category with different characteristics than the present ones that can be suitable as an event store.

At last, it also has to be said that a relational database fits very well for being used as an event store. It meets all the requirements on an event store and it is easy to make queries for the needed operations. Therefore it could be interesting to evaluate other relational databases.

Bibliography

[1] Sandthorn event sourcing. `https://github.com/Sandthorn`, 2015.

[2] Martin Fowler. Event sourcing. `http://martinfowler.com/eaaDev/EventSourcing.html`, 2005.

[3] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. *In Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 14–12, 2013.

[4] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[5] Dominic Betts et al. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure.* Microsoft patterns & practices, 2013.

[6] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[7] Vaughn Vernon. *Implementing domain-driven design.* Addison-Wesley, 2013.

[8] Greg Young. Cqrs documents by greg young. `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`, 2010.

[9] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 7–10. ACM, 2000.

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

[11] Dan Base. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[12] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[13] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2012.

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

[15] Eric Evans. Nosql: What's in a name? `http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html`, 2009.

[16] Giuseppe DeCandia et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[17] Db-engines ranking. `http://db-engines.com/en/ranking`, 2015.

[18] Neo Technology. The neo4j manual v2.2.2. 2015.

[19] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O'Reilly Media, Inc., 2013.

[20] David Montag. Understanding neo4j scalability. *White Paper*, 2013.

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Johan Rothsberg