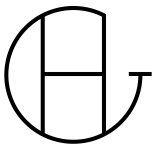

HoGent

Faculteit Bedrijf en Organisatie

# Comparative Study of NoSQL Data Storage Solutions for a social Recent Activity Feed

Florian Dejonckheere

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Chantal Teerlinck
Co-promotor:
Guy De Tré

Instelling: Open Webslides

Academiejaar: 2017-2018

Tweede examenperiode

Faculty of Business and Information Management

# Comparative Study of NoSQL Data Storage Solutions for a social Recent Activity Feed

Florian Dejonckheere

Thesis submitted in partial fulfilment of the requirements for the degree of professional bachelor of applied computer science

Promotor:
Chantal Teerlinck
Co-promotor:
Guy De Tré

Institution: Open Webslides

Academic year: 2017-2018

Second examination period

# Preface

The idea for this research originally comes from the Open Webslides project and its many little side activities in development. One of the things that has always fascinated me was how the platform would handle a massive influx of users, and specifically how it would relate to the non-critical data storage of the news items in the *Recent Activity* feed. I wanted to find out how a NoSQL data store would be integrated into the flow of data, and what kind of data store would be the most efficient, scalable solution for this problem. My interest in this problem was also piqued by using the Neo4j graph database in a personal project, and how the data of the Open Webslides project would fit into the graph theoretical model as opposed to the relational model. Digging into this subject while still maintaining my vision on the Ruby on Rails implementation in the platform allowed me to let the question bloom into this research paper.

This thesis was in part achieved by the support of Chantal Teerlinck, my promotor, who has given me many tips and tricks, and provided a framework for conducting a proper research. Guy De Tré, my co-promotor, also had an important influence on decisions taken in the research and development phase, being a person who is immersed in the academic world of relational and non-relational database management systems. Finally, my friends and family also deserve recognition for helping me accomplish this paper, which is the culmination of three years higher education in a fast-moving and innovative field.

Florian Dejonckheere, Ghent, May 2018

# Samenvatting

De opkomst van grootschalige, dynamische web applicaties heeft geleid tot een enorme toename in de behoefte voor performante database systemen om de overvloed van gegenereerde informatie op te slaan. Het antwoord van de industrie op dit probleem is de NoSQL beweging, die beschikbaarheid en schaalbaarheid verkiest over belangen zoals data consistentie en betrouwbaarheid. Traditioneel richten relationele database management systemen zich meer op de laatstgenoemde belangen. Een belangrijker wordende vraag voor onderzoekers en ontwikkelaars in het vakdomein is hoe de opslag van deze informatie efficient kan gebeuren, om de performantie van de queries te maximaliseren, toegepast op de relevante use case en inherente structuur van de betrokken data.

Deze thesis duikt in de wereld van NoSQL en niet-relationele data modeling door middel van een vergelijkende studie van diverse NoSQL data store categorieën en leveranciers. De *Recent Activity feed* van het Open Webslides project wordt gebruikt als case study. Een vergelijkende studie voor drie NoSQL data stores wordt naar voren geschoven, waarbij factoren relevant voor de use case besproken worden. Vervolgens worden twee logische datamodellen voor document en graph data stores opgesteld, samen met twee bijbehorende implementaties voor de MongoDB en Neo4j NoSQL data stores.

Het onderzoek concludeert dat document stores de meest efficiënte oplossing bieden onder de vergeleken categorieën. De NoSQL data stores worden in het Open Webslides platform gebruikt complementair aan een relationele database, gebruik makend van *polyglot persistence* om een performante en schaalbare applicatie te verkrijgen.

De NoSQL implementaties ontwikkeld in deze thesis zullen bekwame en krachtige oplossingen bieden voor het probleem voorgesteld in de case study. Er zijn echter ook veel mogelijkheden om de grenzen van dit onderzoek te overstijgen boven de use case.

# Abstract

The advent of large scale, dynamic web applications has led to a massive increase in the need for performant database systems to store the deluge of generated information. The industry's answer to this problem is the NoSQL movement, which prioritizes availability and scalability over concerns such as data consistency and reliability. Traditionally, relational database management systems focus more on the latter concerns. An increasingly interesting question for researchers and developers in the field is how to store this data in order to maximize the query performance, considering the use case and the inherent structure of the data involved.

This paper dives into the world of NoSQL and non-relational data modeling, comparing and examining the different NoSQL data store types and vendors. In this comparative study, the *Recent Activity* feed of the Open Webslides platform is used as a case study. For three NoSQL data stores, a comparative study is presented in which aspects relevant to the use case are compared. Subsequently, two logical data models for document and graph data stores are designed, along with two corresponding implementations for the MongoDB and Neo4j NoSQL data stores.

The research concludes that document stores are the most efficient data stores among the solutions considered. The NoSQL data stores are to be used complementary with a relational database, leveraging *polyglot persistence* to achieve a performant and scalable web application.

The NoSQL implementations developed in this thesis will provide capable and powerful solutions for the problem presented in the case study. However, it was also found that there are many opportunities to extend this research beyond the initial use case.

# Contents

# List of abbreviations

**BASE** Basically Available, Soft state, Eventual consistency. 42

**CAP** Consistency, Availability, Partition tolerance. 42

**MVCC** Multi-Version Concurrency Control. 42

**RDBMS** Relational Database Management Systems. 29

# List of Figures

# List of Tables

# 1. Introduction

## 1.1 Context

Up until 500 years ago, knowledge was transferred mainly through verbal communication. Only later were written records and publications added as a method of knowledge transfer. Because of recent advances in technological constraints, course material in the 21st century covers much more ground related to the way knowledge is stored. Modern courses consist of slides, videos, websites and interactive applications. These novelties complement the classical course texts, and are valuable additions in order to support various didactical principles (Cottenier et al., 2016). Allowing students to learn the same content in different ways stimulates the encoding of the content on the relevant parts of the brain (Paivio, 1969).

However, there is a lot more potential to gain from the modernization of educational content, both for the teacher and the students. Education is still too often a one-way street, where students are obligated to process the course content without being provided much challenge or activity (Open Webslides, 2017a). This does not allow for any dialogue to take place between students and teachers concerning feedback and improvement of the course material itself.

Current educational software solutions do not allow co-creation discourse between teacher and students easily, in many cases due to technological constraints („Co-created Courses through Open Source initiatives", 2018). Course material being locked to specific versions of proprietary software is just one of the many problems teachers might encounter when trying to apply this concept in real life. Using interactive tools and applications, the teacher can engage the students more directly.

By building on modern, open standards, the Open Webslides project aims to provide a platform that solves these problems (Open Webslides, 2017a). It creates a user-friendly environment where teachers can create courses based on open source technologies and standards, and it allows teachers and students to apply the co-creation narrative easily. This also enables users to share their material not just with their immediate environment but with a much broader educational audience.

## 1.2 Problem statement

The Open Webslides platform incorporates several ways to stimulate spontaneous co-creation between teachers and students. One of the most prominent elements is the *Recent Activity* feed. This reverse chronologically ordered list enumerates the most recent user interactions with the platform and with other users. Feed items range from simple actions such as a user having created or modified course content, to more complex social interactions like a discussion facilitated by comments or a student's changes being incorporated in a teacher's courses.

The size of the Recent Activity feed data set is directly correlated to the size and activity of the users. The Open Webslides platform is built to cater to higher education institutions. This entails that the traffic on the platform will be relatively high yet predictable, and seasonally bound. It has the potential to grow explosively in timespans critical to teachers and students, such as examination periods. In order for the infrastructure to be able to handle the deluge of the retrieval requests, designing a system that allows for efficient querying and easy scalability is of paramount importance. Further decoupling of this subsystem from the business-critical processes is also important to ensure that downtime of this subsystem has no impact on round the clock availability of course content to the users.

As a result, the Open Webslides team is looking for an efficient solution to this problem, in the form of a NoSQL data store integrated into the platform. It is important to note that this NoSQL data store will complement the relational data store already present in the platform. It does not contain business critical data and it is not an authoritative source of information. The use of multiple data stores characterised by different data models is called polyglot persistence (Sadalage & Fowler, 2012).

This research thesis will provide a comparative and empirical study of existing NoSQL database solutions for the Open Webslides project to implement an efficient, scalable data storage system in the context of the *Recent Activity* feed. A basic solution is already implemented within the platform, however the proposed data model allows for more flexibility and accommodates any future functionality expansion.

## 1.3 Research questions

The research in this thesis is focused on finding an answer to three main research questions:

1. What frameworks and software packages currently exist in the industry to store structured non-relational graph or document data and how do these data models differ from each other?
2. How is the social graph as introduced by the Open Webslides' Recent Activity feed conceptually and logically structured and how is this data consumed?
3. What NoSQL data store is the most appropriate and efficient data store to store this social graph?

Determining and exploring the NoSQL database landscape will provide us with a general idea of the current state of affairs. This knowledge will then be utilized to answer the second research question in form of logical data models, respective physical implementations and reference queries. Finally, The first two questions will then lead into an empirical study to answer the last research question. It will also provide a practical approach for the Open Webslides development team to take into account this research paper when developing the platform in the future.

## 1.4 Research goal and objectives

This research consists of two panels. The first is a comparative study of existing NoSQL products applied to the Open Webslides use case. This chapter may be of interest to a broader audience and future researchers analyzing similar use cases. Second, a concrete recommendation of a NoSQL data store for the Open Webslides project will be made, together with a reference implementation.

## 1.5 Expected results and conclusions

We expect to find a comprehensive answer to all of the research questions proposed in this chapter. Primarily, this involves finding the best NoSQL data model for the studied use case. We expect to find that graph databases are the most fitting data store in this context. This results from the train of thought that the Open Webslides Recent Activity feed data is structured as a directed graph, and behaves as such.

Furthermore, since physical implementations will be developed during this research in order to perform the empirical study, the main result of this thesis is expected to be a concrete recommendation for a NoSQL product, and a reference implementation of the Recent Activity feed in the respective NoSQL product.

In chapter 2 an overview will be presented of current research, applied solutions and other related work in the research domain based on a literature study. This literary review will then be used to give a broad and global overview of the research domain, introducing

various relational and NoSQL concepts and explaining the ideas behind NoSQL and polyglot persistence in chapter 3.

Chapter 4 will clarify the methodology used in this thesis to attempt to formulate an answer to the research questions. The research is split into three distinct parts. The first part, chapter 5, will evaluate existing NoSQL data stores using a comparative study in the context of the Open Webslides use case. Certain NoSQL categories will also be ruled out in this chapter, and a concrete selection of NoSQL database management systems will be made for further comparison.

The second part in chapter 6 describes the domain model that is used in the Open Webslides project, and proposes the logical and physical data models for every included NoSQL database. Reference queries that cater to the needs of the domain will also be drawn up and implemented. Chapter 7 combines the proposed data models and reference queries in order to perform and discuss an empirical study on the data stores selected in chapter 5.

Chapter 8 will conclude and reflect upon the opportunities and inadequacies encountered in this paper, providing an entrypoint for future research.

Finally, chapter 9 will present a general conclusion and summarize the research, formulating answers on all of the research questions.

# 2. State of the Art

Ever since the rise of the NoSQL databases in 2009 (Sadalage & Fowler, 2012) it has been a subject of vigorous academic and professional research. The contrast with relational databases, optimal use cases, performance and scalability are only some of the aspects that have been analyzed with great regularity. This chapter will summarize previous publications relevant to this thesis.

The book by Sadalage and Fowler (2012) provides an excellent entry into the world of NoSQL. It explains the motivation behind the use of NoSQL techniques, and how this differs from relational data storage. Furthermore it introduces the segregation of NoSQL data stores into four main categories: key-value, document, column and graph databases. Second, Sadalage and Fowler touch the concept of polyglot persistence. This describes the concept of an application using multiple types of data stores to store heterogeneously structured data. This technique is relevant in particular to this thesis, as the described data schema only relates to one of the database management systems integrated in the Open Webslides platform. The second part of the book provides a more practical approach to using polyglot persistence in an enterprise application. The authors have written down many pointers and guides in order to pick the right database for a particular use case.

There have already been numerous studies to differentiate the different types of NoSQL databases and comparative studies between NoSQL database systems. Nayak, Poriya, and Poojary (2013a) present a fifth NoSQL category: object-oriented databases. Other studies such as Moniruzzaman and Hossain (2013) and Maroo (2013) provide a feature-based comparison for various NoSQL vendors and database systems. Finally, the similarity and resemblance of relational and NoSQL data stores is also a well-researched topic in current literature. Studies and surveys such as Mohamed and Ismail (2014) and Cattell (2010) tackle this subject in great detail.

Grolinger, Higashino, Tiwari, and Capretz (2013) present a use-case based approach to comparing different NoSQL and NewSQL data stores. The survey incorporates a feature-based comparison over different aspects such as querying, scalability and security, and analyzes these concepts in the context of a select number of NoSQL data stores.

Hecht and Jablonski (2011) provides a feature-based comparison of different NoSQL database types and vendors. The researchers compare the data model, querying access, concurrency, partioning and replication. The paper uses a duality-based approach, where a minus indicates that the feature is not supported by the database system, and a plus if the feature is supported. The paper also presents the problem of a lack of unified querying interface for NoSQL databases. Furthermore, the importance of choosing the right NoSQL database type for the use case is emphasized, however Hecht and Jablonski do not present a specific case study.

The proceedings of the 2013 IEEE International Conference on Big data by Kaur and Rani (2013) describe the theoretical modeling and querying of SQL and NoSQL data stores. The paper then proceeds with a case study of a social networking site similar to Slashdot (Malda & Bates, 1997). Starting from an entity-relationship diagram (ERD), the researchers then proceed by modeling the entities in both a document and a graph database. Finally, a set of seven queries related to the use case is then drawn up and compared for the PostgreSQL, MongoDB and Neo4j data stores.

Zhao (2015) explores the use of NoSQL data stores to store huge amounts of observational data generated by astronomical research. It briefly discusses using filesystems and relational data stores, before comparing NoSQL alternatives. A concrete data model to store the astronomical data in a MongoDB data store is then presented, together with eight scenarios and queries that may be used in a production system. Furthermore, performance measurements of MongoDB are also analyzed. Data insertion, querying and deletion using the aforementioned data scheme and real observational data are used in this section.

The proceedings of the AGILE 2015 conference by Schmid, Galicz, and Reinhardt (2015) present an overview of selected SQL and NoSQL databases, focusing on the geo-functionalities of the systems. It uses performance tests between two document-based NoSQL data stores (MongoDB and CouchBase). The researchers conclude that geospatial calculations in NoSQL database systems are still only supported for basic queries. Relational databases still perform superior to NoSQL databases in small to larger data sets for queries with geo-functions. However the NoSQL response time only increases slightly relative to data set size.

The technical report by Barahmand, Ghandeharizadeh, and Li (2015) quantifies the scalability of MongoDB and HBase for processing simple operations using the social networking benchmark BG (Barahmand & Ghandeharizadeh, 2013). It considers both horizontal and vertical scalability of the data stores using the Social Action Rating (SoAR) introduced by the benchmarking tool. In order to perform these benchmarks, two logical data models for the database design are presented. The report concludes that while both data stores scale superlinearly, their speedup is limited by the resources of a few nodes out of many becoming fully utilized.

Another performance-based study written by Abramova, Bernardino, and Furtado (2014) compares five popular NoSQL databases (Cassandra, HBase, MongoDB, OrientDB and Redis) using the Yahoo! Cloud Serving Benchmark (**Yahoo2010**). The study compares read and write query performance. It concludes that over the five compared data stores, MongoDB, Redis and OrientDB are more read-optimized, and Cassandra and HBase are more update-optimized.

A more query-oriented study was performed by Zhou, He, Sheng, and Wang (2013). The paper considers both the academic and the industry definition and description of data models and system architectures. The researchers identify two kinds of searching approaches: the MapReduce-oriented and the SQL-like querying.

The work of Atzeni, Bugiotti, Cabibbo, and Torlone (2016) dives deeper into the world of non-relational data modeling. The paper investigates how traditional data modeling can be used in the context of schemaless and heterogeneous data stores. Atzeni et al. propose NoAM (NoSQL Abstract Modeling), an abstract data model to describe NoSQL databases based on the common surfaces of the various data store types. This technique can be used to describe system-independent application data and later to implement this in the specific data stores, taking advantage of the various target system idiosyncrasies. Further articles by the same authors (Bugiotti, Cabibbo, Atzeni, & Torlone, 2014) expand upon this abstract data model to present a database design methodology for NoSQL systems.

The main differences between this study and the previous studies are:

1. Many studies have been conducted to understand the motivation between the NoSQL principles and the shift from relational data stores. The division of NoSQL data store types into four most commonly recognized categories and elaboration upon this is usually also a topic in these studies. This research paper builds upon that knowledge, providing only a brief introduction in the world of NoSQL and NewSQL concepts.
2. Some of the previously mentioned research papers also discuss a case study applied to a specific use case. This is mostly related to business critical systems that store and process large volumes of data. The use case described in this thesis is very specific in that it's a complementary subsystem that does not affect critical data. Consequently, certain comparative attributes such as security and availability are not considered in this research.

This thesis aims to provide a case study of data storage the Open Webslides (2017a) platform. Several use case based surveys and studies already exist, however they aim at replacing a relational database in an application with a NoSQL database without bringing polyglot persistence into account. Sadalage and Fowler (2012) is one notable exception in this aspect. In the case of Open Webslides, the NoSQL data store only complements the relational database and does not fulfill a critical function. Therefore, several constraints such as security and availability differ in interpretation from existing studies.

# 3. Overview

## 3.1 Relational data stores

Relational Database Management Systems (RDBMS) have been the de facto standard for over two decades to efficiently store and query information in a wide variety of native and web applications. These data stores are based on the relational model devised by Edgar Codd (Codd, 1970). The data in the system is presented as relations: a collection of data consisting of rows and columns. The user of the database can query and manipulate the data using relational operators.

Codd presented thirteen rules for a database in order for it to be considered a *relational database* (Codd, 1985). However, many of the modern database systems do not adhere strictly to all of these rules. More commonly, a relational database is defined as a database that exposes its information using a collection of rows and columns.

Relational database management systems provide certain guarantees during database transactions. Härder and Reuter discussed recovery in transaction-oriented databases, and introduced the concept of ACID: an acronym referring to a set of transactional properties (Härder & Reuter, 1983).

1. **Atomicity**
   Each transaction is "all or nothing", either the transaction completes successfully and the data is mutated in an atomic way, or the transaction fails in its entirety and none of the data in the transaction is committed to the database.
2. **Consistency**
   Each transaction, when successful, only commits *legal* results. This means that data in the transaction and subsequently in the database does not violate any constraints.

The database is always in a consistent state.

3. **Isolation**
   Actions within a single uncommitted transaction are not visible to other, concurrently running transactions. Once the transaction has successfully completed, the data is visible for the other transactions.
4. **Durability**
   Once a transaction has completed successfully and been committed to the database, the system must guarantee that these results survive any subsequent malfunction.

## 3.2 NoSQL data stores

By 2009, a totally different concept of data storage was popularized (Leavitt, 2010). NoSQL data stores provide a system of storage that is "non SQL" or "non relational" (The NoSQL Archive, 2018). Furthermore, properties of NoSQL data stores included horizontal scalability, inherently distributed and open source. More recently, the NoSQL denomination has also been explained as "not only SQL", pointing on the fact that most NoSQL data stores provide a different interface besides SQL. Many databases expose a REST API as primary execution interface. Other data stores have devised their own binary communication protocol, such as the Bolt protocol (Neo Technology, 2007a) for the Neo4j graph data store.

Another concept that became popular together with the increase in dataset size is MapReduce. MapReduce is a conceptual programming framework and physical implementation for processing big data sets using multiple, distributed nodes (Dean & Ghemawat, 2008). Some NoSQL data stores support this paradigm natively, while others have later added support for it.

The use of non-relational data stores was motivated by the needs of Web 2.0 companies such as Facebook and Google (Mohan, 2013). NoSQL provides a way to store and process massive amounts of data in a flexible way. The architecture of such systems is usually more simple than the equivalent relational database systems, and are more aimed at improving horizontal scalability as opposed to vertical scalability. Data is not stored in rows and columns as is the case in the relational model, but rather in a different data structure.

Sadalage and Fowler (2012) reject the proposition that NoSQL data stores replace relational data stores. Rather, the technology is meant to complement the relational one, and substituting one for another is not deemed to be a potential solution for performance issues. Since both systems are designed from the ground up with very different ideas in mind, developers have to think about the potential advantages and pitfalls of each. NoSQL has certain use cases where it shines, whereas the relational data model is a much better fit for other purposes. The use of multiple data storage technologies and database management systems within the same application is called *polyglot persistence*.

Nayak et al. (2013a) divide the data models used by NoSQL data stores into five categories. The following sections describe these five categories, and present three additional categories

that can be identified in recent NoSQL database trends.

### 3.2.1 Key-value

Key-value data stores are simple in design, yet powerful and efficient when used in the right circumstances. A key-value data store allows the user to store schemaless data using an opaque, unique key, creating a *key* and *value* pair. The values are stored in a manner similar to hash tables or dictionaries commonly found in programming languages and standard libraries. Queries are processed by looking up the value for the provided key, which is used as an index in the database. Modern key-value data stores prefer high scalability over consistency, resulting in the fact that more advanced ad-hoc querying and analytical operations on the data such as joins are not supported.

### 3.2.2 Document

Document databases store their data in *documents*, indexed by a unique key. Documents are usually structured in a hierarchical manner and represented in the JSON format. Document stores are technically a subclass of key-value stores, however the difference lies in the interpretation of the data itself. In contrast with key-value data stores, the value (document) is not opaque to the database management system, but is parsed and interpreted, and subsequently used for query optimization. Some document data stores may provide advanced query capabilities on the contents of documents. Since documents are schemaless, each document may have a similar structure, or a completely different one.

### 3.2.3 Column-oriented

Column-oriented data stores are designed to store data by column rather than by row. This kind of NoSQL data store is more similar to relational database systems than the other categories, in regard to data structure used to store the data (Daniel J. Abadi, Madden, & Hachem, 2008). In column-oriented data stores, each key is associated with a set of attributes, stored in columns. Concretely this means that the data is indexed by column value, rather than by row. Column-oriented data stores are commonly used for queries where only a subset of the attributes is retrieved, as opposed to row-based data stores where the entire row is returned, after possibly discarding any unused values (Daniel J Abadi, Boncz, & Harizopoulos, 2009).

### 3.2.4 Graph

Graph data stores keep their data persisted in the form of a graph. The graph is made up of *nodes* and *edges* as the graph-theoretical model describes (West et al., 2001). The former are the database entities that contain the data itself, similar to tables and their respective columns in the relational model. The latter are the relations between these entities. Graph data stores use a technique called *index-free adjacency*, where every node contains a logical

pointer to the adjacent node (Weinberger, 2016). This makes graph traversal a very fast operation. Some graph databases like Neo4j are ACID compliant (Miller, 2013).

### 3.2.5 Object-oriented

Object-oriented data stores represent the data as an object, closely resembling the concept of an object in object-oriented programming languages. This puts object-oriented data stores much closer to the programming environment than other database systems. It provides all features inherent to object-oriented languages, such as data encapsulation, inheritance and polymorphism. The concepts of class, class attributes and an object can be mapped onto the relational concepts of a table, columns and a tuple. This concept of data storage follows the programming model much better and makes software development more flexible.

### 3.2.6 Multi-model

Data stores are generally built and optimized around one data model. However, databases supporting multiple data models exist as well. These database systems allow storing data using any of the different data models mentioned before, while integrating these into the same server package. One example of such a data store is OrientDB (OrientDB Ltd, 2010). Multi-model data stores support and facilitate the principle of polyglot persistence while reducing operational complexity of running multiple database management systems.

### 3.2.7 NewSQL

NewSQL is a type of relational database management system that aims to provide the same scalability and distributed performance of NoSQL data stores (Grolinger et al., 2013). NewSQL data stores are ACID compliant.

### 3.2.8 Triple store

A triple store is a type of data store similar to key-value and graph data stores. Triple stores process data using semantic queries on data triples. A triple consists of a *subject*, an *predicate* and an *object* (Rohloff, Dean, Emmons, Ryder, & Sumner, 2007).

Most NoSQL data stores are built around the concept of *eventual consistency* (Brewer, 2000). This is a consistency model that dictates that all accesses to a particular piece of data will eventually return the last updated value. This principle is broadly implemented in distributed computing systems. Systems providing this property are also classified as BASE: Basically Available, Soft state, Eventual consistency. In contrast to the ACID properties, systems built around the BASE principles prefer availability over consistency.

In 2000, Eric Brewer presented a conjecture known as the CAP theorem (Brewer, 2000). This conjecture, later formally proven (Gilbert & Lynch, 2002a, 2), asserts that it is impossible for a distributed data store to exhibit more than two out of three of the following properties:

1. **Consistency**: Every read operation receives the most recent write result
2. **Availability**: Every request receives a non-error result
3. **Partition tolerance**: The system continues to work despite failure to communicate between nodes

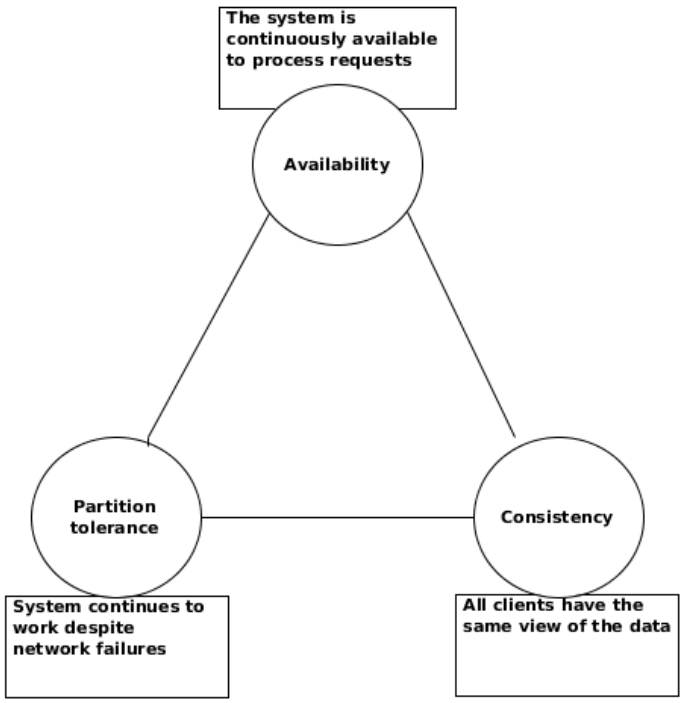


Figure 3.1: CAP Theorem

The CAP theorem states that when a network partition is present, the database developer has to choose between providing consistency or availability. Note that *consistency* as defined by the CAP theorem is not the same concept as consistency as described by the ACID properties. Database systems respecting the ACID guarantees choose consistency over availability, while database systems built on the BASE principle generally choose availability over consistency.

# 4. Methodology

Analyzing and comparing every NoSQL data storage solution is not feasible due to the sheer number of competing products. Therefore only a selection of the most popular NoSQL data stores in the different NoSQL categories are considered in the research. This selection is motivated by a list of the most popular databases kept up to date by Solid IT (2018), and by the Gartner Magic Quadrant 2018 (Gartner, Inc., 2018a). For every category, we select the most popular databases by quadrant and reported score, which is in turn based on several other criteria. DB-Engine Ranking attempts to estimate the overall popularity of the data store on the Web.

Next, the selected data stores are subject to a use case based comparison. In this comparative study every solution is analyzed based on various aspects. The data model, which is the main driving force behind many other aspects, is one of the main focus points. The NoSQL data stores are also discussed in relation to querying capabilities, scaling, partitioning and replication, consistency and concurrency control. Certain aspects are not analyzed in detail due to their irrelevance to the use case, such as database security, authentication and auditing capabilities.

In the subsequent chapter, the conceptual and logical data model of the Open Webslides project is presented. Building upon this, a physical data model is discussed and implemented for the various NoSQL data stores that were selected in the first part of the paper. Next, the typicial data access flow within the application is examined, and five reference queries are introduced. These queries are examples of queries that could typically be used to retrieve data from the data store as part of the normal operating procedures of the Open Webslides application.

Finally, the implementation of the data model and the reference queries are used in the last part of the research as a base for qualitative benchmarks. Since the Open Webslides application is built on Ruby and Ruby on Rails, the benchmarks will be implemented in a Ruby on Rails application, making use of the available Ruby language bindings to the various data stores examined. For every data store used in the benchmark, the most popular ORM gem is used as determined by the RubyGems repository (RubyGems, 2003). Related work and previous NoSQL database benchmarks are also considered in this part, however it is referenced only as a baseline due to the fact that previous work does not cover the specific use case and NoSQL data stores studied in this research.

# 5. Data stores

## 5.1 Selected data stores

Due to the large number of active and maintained NoSQL data stores it is not feasible to consider all of them for this research. Every NoSQL category described in chapter 3 will be considered, discussing the use of the respective categories applied to the use case, and decided upon whether or not it will be included in the research. In the applicable NoSQL categories the most popular data stores are selected, where popularity is based on certain predetermined parameters.

Solid IT (2018) maintains a list of database systems ranked by popularity, based on parameters such as number of mentions on websites, Google Trends and relevance in social networks. These parameters are also cross-referenced with professional networks such as LinkedIn (Microsoft Corporation, 2002) and Upwork (Upwork Global Inc., 2015) using the number of available job offers and professional profiles. This ranking is called the DB-Engine Ranking. The list includes not only NoSQL databases but also other types of data storage systems such as relational database systems.

The information technology division of Gartner maintains a yearly report on emerging relational and non-relational database technologies (Gartner, Inc., 2018a). The Gartner Magic Quadrant is a yearly or bi-yearly market research report that summarizes market trends in many technological sectors. Gartner uses proprietary methods to qualitatively score vendors on completeness of vision and ability to execute (Gartner, Inc., 2018b). Vendors are categorized in four quadrants: *leaders*, *challengers*, *visionaries* and *niche players*.

The DB-Engine Ranking is used primarily to determine which data stores are the most

interesting to consider in this research. The Magic Quadrant for Operational Database Management Systems was used as a secondary resource. Due to licensing concerns regarding the Open Webslides project, database systems that do not fall under a free license as classified by the Free Software Foundation (1985) will not be considered.

Certain categories of NoSQL data stores will be omitted from the comparative study. Key-value stores will not be accounted for due to the relative simplicity of this type of data store. The main use case of key-value stores is not storing more complex information, rather the emphasis lies more on scalability and consistency. Processing complex queries that consist of the NoSQL equivalent of relational joins is not efficient in these data stores. Implementing the proposed data model and queries is a more ambitious task that is not aligned with the scope and goals of the research.

Column-oriented data stores are a category of NoSQL data stores that is questionably useful in this case study. These data stores are commonly seen as inverse relational database systems, where the storage of attributes per entity is more flexible regarding nullable values and unstructured information (Daniel J Abadi et al., 2009). Column-oriented data stores are very efficient when retrieving a subset of columns for a certain record. Since the data store will be deployed as additional data store next to the authoritative relational database, it will not contain any superfluous information that will not be used when querying the database. This workflow cancels out the efficiency and usefulness of column-oriented data stores, and subsequently this NoSQL category will not be considered as a viable candidate.

Furthermore, comparison and application of NewSQL data stores will not be included in this paper either. As described in chapter 3, NewSQL aims to provide scalable performance similar to that of NoSQL while still guaranteeing ACID properties. Since ACID is not a major concern for this use case, NewSQL does not provide significant advantage over NoSQL, and will therefore be omitted from the comparison. Similarly, object-oriented and multi-model database such as OrientDB (OrientDB Ltd, 2010) are not within scope of this research.

In conclusion, document and graph data stores will provide the most beneficial data storage and are the main focus of this research.

The most popular document database systems according to the DB-Engine Ranking are MongoDB (MongoDB Inc., 2009a), CouchDB (Apache Software Foundation, 2005b) and Couchbase (Couchbase, Inc., 2010). While Couchbase is technically a multi-model data store, it is being ranked as a document database. However, Couchbase is a conceptual merge between CouchDB and Membase, and adds improved clustering capabilities and strong consistency guarantees (Couchbase, Inc., 2018). Couchbase will not be investigated upon in a practical capacity, however it is considered as an option to increase multi-host scalability. This leads to MongoDB and CouchDB being the contestants in the document database system category.

Finally, as sole graph database the Neo4j system (Neo Technology, 2007b) stands out over competitors in the ranking. This database system will be analyzed as graph database candidate in this research.

The data stores included in the comparative study will be MongoDB and CouchDB as document databases, and Neo4j as graph database.

## 5.2 Comparative study

Data stores and databases can be compared and analyzed using many quantitative and qualitative aspects. In this section we propose a selection of criteria based on the usefulness applied to the studied use case. Since the landscape and feature-set of NoSQL data stores is changing on a weekly base, the impact of these technologies must be carefully considered in order to reach a durable conclusion in this research.

The features of a data store that are taken into account in this comparative study are:

- Querying capabilities
  - Language
  - Protocol
  - MapReduce
- Programming language
- Language bindings
- Integrity model
- Atomicity
- Revision control
- Consistency
- Persistence
- Partitioning
- High Availability
- Concurrency
- Replication
- License
- Commercial support

Certain aspects are not relevant to the presented use case because of various reasons. Aspects omitted from the study are:

- Security
- Compression
- Full-text search
- Geospatial functionality
- Cloud hosting

The specific use case described in this thesis does not focus on security, since the data store is not public facing and clients do not interact directly with it. Security measures

include but are not limited to authentication, authorization, encryption and auditing. None of these are features that are required or useful for the comparison. Authentication and authorization is functionality that is present in all databases. It introduces the concept of multiple clients or roles connecting to a database, and assigning permissions to these clients in order to enforce permission-based access. However since there will only be one client connecting to the database - the platform itself - deeper integration with LDAP, ActiveDirectory or similar is superfluous and omitted from the comparison.

Encryption refers to the mechanism where data is encrypted and unreadable for unauthorized third parties. Encryption in databases is threefold: encryption of data at rest, client-to-server communication and server-to-server communication (Grolinger et al., 2013). However, since data protection is a comprehensive topic, it does not fall within the scope of this research. Consequently, encryption functionality is not considered in the comparative study.

Database auditing is a facility offered by the database management system that keeps track of the usage of database resources and authorization. Operations on the database leave a trail of events, called an *audit log*. Similarly to authentication, the usefulness of this functionality is somewhat lost when there is only one client operating on the database. However, many security standards such as PCI-DSS and HIPAA require the existance of an audit log.

Compression of data in the database is not included in the comparison. Builtin compression may provide additional storage space but as it is a disk space-CPU usage tradeoff we have chosen to only consider CPU usage. Akin to compression, we leave the choice of cloud hosting up to the database administrator.

Full-text search and geospatial functions are provided by certain data stores, however these features are not used by the platform and subsequently are not relevant to this comparative study.

| | | Querying | | | Language | |
|---|---|---|---|---|---|---|
| | | **Language** | **Protocols** | **MapReduce** | **Language** | **Language bindings** |
| **Document stores** | **MongoDB** | JavaScript | MongoDB Wire Protocol<br>REST [1] | Yes | C++ | C/C++, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala<br>Erlang[1], Go[1] |
| | **CouchDB** | REST | HTTP | Yes | Erlang | C/C++[1], Dart[1], Go[1], Java[1], Lua[1], Node.js[1], Python[1], R[1], Ruby[1], Scala[1] |
| **Graph stores** | **Neo4j** | Cypher<br>SparQL[1]<br>Gremlin[1] | HTTP<br>Bolt | No | Java | C#, Java, JavaScript, Python, C/C++[1], Clojure[1], Erlang[1], Go[1], Haskell[1], Perl[1], PHP[1], R[1], Ruby[1] |

Table 5.1: NoSQL data stores: Querying capabilities and language support

[1] 3rd party community supported

One of the most important factors when deciding on a NoSQL data store is the capability to communicate with the data store, since NoSQL does not have a unified interface like SQL for relational databases. Choosing a certain data store may allow the developer to integrate data storage easier into the application. Every NoSQL data store has its own standard for composing queries, related but not depending on the protocol used for communication with the server. MongoDB allows querying using a JavaScript API, natively over MongoDB's binary protocol or over HTTP (REST) using a third party plugin. Subsequently, MongoDB is popular among server-side applications written in JavaScript and Node.js (Dayley, 2014). CouchDB takes another approach and provides a RESTful interface over HTTP to query, modify and manage the database.

The Neo4j graph store is a different story. The native querying language of Neo4j is called Cypher, and is much closer to SQL than MongoDB or CouchDB's way of querying. Cypher can be used natively both over HTTP and over Neo4j's binary Bolt protocol. Third party plugins can extend Neo4j to provide interfaces for SparQL and Gremlin querying languages.

Another factor that might come into play when choosing a data store is the ability to support MapReduce, explicitly or under the hood. MapReduce is a conceptual framework and implementation for processing large data sets using multiple processing entities (Dean & Ghemawat, 2008). It is composed of a *map* and a *reduce* procedure. The former filters and sorts the data, while the latter performs an aggregating or summarizing operation as a result to the query.

The chosen document data stores both support MapReduce as normal operating procedure. The graph store does not support Neo4j, instead building upon *index-free adjacency*.

Luckily, all major vendors prove to be adequately supported by first- or third-party efforts.

| | | Integrity | | | |
|---|---|---|---|---|---|
| | | **Model** | **Atomicity** | **Revision control** | **Consistency** |
| **Document stores** | **MongoDB** | BASE | Document level | No | Configurable:<br>Eventual or strong consistency |
| | **CouchDB** | BASE | Document level | Yes | Eventual consistency |
| **Graph stores** | **Neo4j** | ACID | Transaction level | No | Eventual consistency |

Table 5.2: NoSQL data stores: Data integrity

Even though all of the compared data stores are classified as NoSQL and based on the principle of *eventual consistency*, the integrity model is different for each. MongoDB is designed with the Basically Available, Soft state, Eventual consistency (BASE) integrity model in mind, which prioritizes availability over consistency over data in the Consistency, Availability, Partition tolerance (CAP) theorem. CouchDB is designed around the BASE model as well, however the data store solved the problem in a different way. Documents in CouchDB are stored using a technique called Multi-Version Concurrency Control (MVCC). Neo4j however, enforces ACID guarantees as only data store included in the comparison.

Data stores designed around the BASE model typically do not provide any strong consistency guarantees. This is reflected in the fact that MongoDB and CouchDB provide atomic operations only on document level. This means that operations on documents – and embedded child documents – are atomic, however operations on multiple documents are not guaranteed to be completely atomic. Neo4j is ACID-compliant and does provide atomic transactions for multiple operations akin to the database transactions found in relational data stores.

CouchDB's concurrency control method is based on the technique of MVCC. Subsequently it is the only data store that provides native version control as a result of the concurrency control method used in the database engine. Neither MongoDB or Neo4j provide any similar feature natively.

NoSQL data stores are in general based on the principle of eventual consistency, as tradeoff versus the availability according to the CAP theorem. However, since the start of the development on MongoDB a lot of research and implementation has been done, and the software will soon provide both the option to configure the database to enable strong consistency, and atomic operations on multiple documents – similar to transactions. Both MongoDB and Neo4j provide eventual consistency as dictated by the principles of the CAP theorem.

| | | Scalability | | | |
|---|---|---|---|---|---|
| | | **Persistence** | **Partitioning** | **Replication** | **Concurrency control** |
| **Document stores** | **MongoDB** | Memory<br>Disk | Shard key | Master-slave | MVCC (document)<br>Locks (global, database, collection) |
| | **CouchDB** | Memory<br>Disk | Consistent hashing | Multi-master | MVCC |
| **Graph stores** | **Neo4j** | Memory<br>Disk | Cache sharding | Master-slave | Locks |

Table 5.3: NoSQL data stores: Scalability

A big part of the featureset that NoSQL data stores have to offer is the possibility to scale horizontally over multiple nodes, called *clustering*. Relational database management systems scale very good vertically on a single node, however scaling to multiple nodes is a more complex issue. This is where NoSQL systems typically stand out, prioritizing availability over consistency according to the BASE principles.

MongoDB, CouchDB and Neo4j all support keeping the dataset in memory, and persistence to disk. Memory accesses are typically magnitudes faster than disk access, independent of whether traditional rotating media or more mordern solid state technologies are used. The database management system will frequently commit the memory pages to disk, to ensure durability of the data.

Since NoSQL data stores are designed from the ground up to be used in a distributed context, the method used to support partitioning is of paramount importance. Distributed databases typically partition networks using a method called *sharding* (MongoDB Inc., 2009b). Sharding is the division of the entire dataset pushed to different nodes based on certain, predetermined criteria. Choosing the sharding criteria in a smart way can allow all sorts of distribution models, from an equal division over all nodes to a weighted distribution based on node capabilities. One technique commonly used by multinational companies is sharding based on geographical location. This ensures that the data of a user in a certain geographical area is pushed to a node close to that geographical area, which in turn helps out server latency and improves the general responsiveness of the server. Facebook is one example of this: the company has a userbase and data centers that stretch over the entire world (Barrigas, Barrigas, Barata, Bernardino, & Furtado, 2015). The algorithm used in Facebook's sharding selection is based on the most important geographical location for a user.

MongoDB sharding is based on a shard key within the document. This shard key can be either determined by the developer to allow more finegrained control over the process, or automatically determined based on the unique identifier assigned to the document in order to allow for a roughly equal distribution. CouchDB's partitioning system works in a different way. It utilizes consistent hashing, which means that the document ID is hashed and the documents are distributed equally over the available nodes (Apache Software Foundation, 2005a).

Finally, Neo4j sharding is based on *cache sharding*, which is aimed at distributing the graph over the nodes in order to send a query that hits a certain area of the graph always to the same node.

Another important feature related to the architecture of a storage cluster is the replication model and capabilities. Replication models are characterised and divided based upon the amount of *masters* and *slaves* in the cluster. A *master* is a server that is the authoritative source for data, while a *slave* is a node that is dependent on the master for certain queries. This implies that in a master-slave replicated cluster, all write requests go solely to the master node, which then replicates the modifications to the slave nodes. While master nodes can handle read and write requests, slave nodes can only handle read requests. The inherent scalability for read requests in this architecture means it is a very good candidate

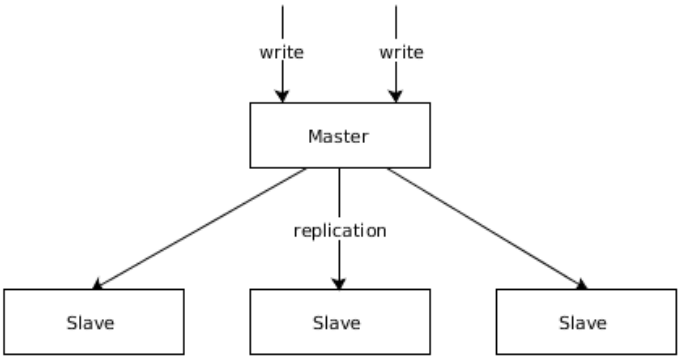


Figure 5.1: Master-slave replication

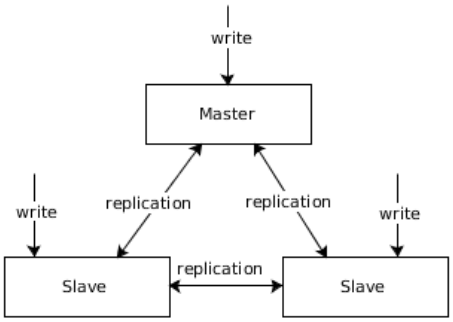


Figure 5.2: Master-master replication

for data models where the read requests outnumber the write requests by a large amount.

As a side-effect, the cluster itself is more resilient to slave node failures: when a slave node goes down, the remaining nodes can balance the load and the cluster can continue to function. However, a master failure in this setup renders the cluster at least read-only, and – depending on the sharding configuration – potentially only able to serve part of the data that was stored on the nodes. Neo4j is an example of a data store that operates under a master-slave replication configuration. In certain cases the cluster is able to automatically recover from a master failure by a process called *leader election*, where a new authoritative master node is chosen (Singh, 1996). This type of capability is commonly seen by data stores supporting more simple data models, such as key-value stores.

Similarly to Neo4j, MongoDB operates in a master-slave configuration: one authoritative master, and an undetermined amount of slave nodes. CouchDB on the contrary, supports a multi-master or master-to-master configuration. These configurations allow for multiple master nodes to exist within the cluster, and CouchDB implements a quorum algorithm between master nodes in order to allow for eventual consistency in the cluster.

Finally, the way a data store implements its concurrency control algorithm is important for both horizontal and vertical scalability. Traditionally, relational database management systems handle concurrent accesses using a pessimistic locking system, utilizing row-, table- and database-level locks (Bernstein, Hadzilacos, & Goodman, 1987). Two type of locks exist in this context: shared locks (colloquially known as *reader locks*), and exclusive

locks (*writer locks*). The former implements a mechanism to allow an undetermined amount of requests to access the data without mutating it, the latter prevents more than one request to modify the data at the same time, provided there are no existing reader or writer locks.

In this aspect, Neo4j behaves more like a relational database. As mentioned before, all write requests are handled by the master node in the cluster, and the concurrent requests are restricted using a lock mechanism.

CouchDB takes an entirely different approach to concurrency control. Revision control is a feature built natively into the data store, and it is used in a mechanism called Multi-Version Concurrency Control (MVCC) to provide concurrent write access to data. In short, MVCC redirects running write requests on a document to a new version of that document, while still serving the old version to concurrent read requests. Once the write request on the new version is finished, the pointer pointing to the most recent version of the document is updated to point to the new, updated document instead of the old, obsolete document. This also has the benefit of automatically making document writes atomic, since only one value has to be updated in the end: the document pointer.

MongoDB also makes use of MVCC, however certain requests are still restricted using locks. This relates to requests that modify the entire database management system, the database or the collection and shared and exclusive locks are enforced respectively.

| | | Features | | |
|---|---|---|---|---|
| | | **License** | **Commercial support** | **Cloud environment** |
| **Document stores** | **MongoDB** | GNU AGPLv3 (database)<br>Apache (drivers) | Yes | Amazon EC2<br>Google Cloud Platform<br>Microsoft Azure<br>Digital Ocean<br>Cloud hosting partners |
| | **CouchDB** | Apache | No | No |
| **Graph stores** | **Neo4j** | GNU GPLv3 (Community Edition),<br>AGPLv3 (Enterprise Edition) | Yes | Amazon EC2<br>Google Cloud Platform<br>Microsoft Azure<br>Digital Ocean<br>Cloud hosting partners |

Table 5.4: NoSQL data stores: Hosting concerns

Table 5.4 provides insight into the aspects related to hosting one or more database instances. First, the type of license is important when hosting the data store. Since we have restricted our research to licenses classified as free by the Free Software Foundation (1985), none of these data stores prevent hosting instances without incurring additional licensing costs. However, for two out of three data stores commercial support is available as well. Commercial support includes licensing the product for enterprise use, which usually includes additional functionality – commonly related to scalability – and customer support.

MongoDB is dual-licensed. The database management system itself is licensed under the GNU Affero General Public License version 3. The language bindings – called drivers – while officially supported are licensed under a different license, the Apache license

The GNU Affero General Public License version 3 is a modified version of the GNU General Public License version 3. The GNU General Public License requires developers that modify software licensed under the GPL to distribute their modifications as well. However, the license does not cover the case of service providers: if a developer modifies the source code and runs the software on a server, allowing other users to interact with it, distribution of the modified source code is not required. The AGPL adds a provision to prevent this loophole. Whenever a modified version of software licensed under the AGPL runs on a server, the modified source code must be available to download. In the case of data stores, this makes sure that the software remains free and cannot be commercially exploited.

CouchDB, being an Apache project, is licensed under the Apache license.

The Neo4j graph store is dual-licensed as well: the Community Edition is licensed under the GPL version 3, and the Enterprise edition is licensed under the AGPL version 3.
Deciding on data store hosting is a difficult topic, which is not within the scope of this research. We merely enumerate the options available for every data store. Since all products are licensed under a free license, it is possible for invididuals and companies to host instances themselves. MongoDB and Neo4j provide commercial support for this use case. However, running a data store instance can also be outsourced to a plethora of companies and hosting providers. Both MongoDB and Neo4j integrate with various cloud providers such as Amazon EC2, Google Cloud Platform or the Digital Ocean hosting platform.

## 5.3 Conclusion

In this chapter the possible NoSQL data storage solutions were discussed using a list of properties applicable to the Open Webslides use case studied in this research. Two document stores and one graph store were selected as interesting candidates in order to implement the Ruby framework in the next chapter. The three data stores – MongoDB, CouchDB and Neo4j – were compared on their most important functional and non-functional properties.

# 6. Data model

In this chapter we describe the conceptual domain of the use case, and provide logical data models applied to the NoSQL data models selected in the previous chapter. Physical data models for MongoDB, CouchDB and Neo4j will be presented, along with an implementation in a demo Ruby on Rails application using Ruby language bindings. Finally, a set of reference queries that may typically be used in the context of the Recent Activity feed will be proposed. These queries will be formally described, and subsequently implemented in the query languages specific to the three data stores.

## 6.1 Domain description

On the Open Webslides platform, no distinction is made between a teacher and a student in the data model. Both are represented by the `User` entity in the database. This entity contains information pertaining to the user, such as email address, first name and last name. The data models described in the next sections will closely follow the existing data model of the platform. However, attributes irrelevant to the Recent Activity feed are omitted from the model and not available in the NoSQL data store, in order to improve efficiency and simplify abstraction.

A user can create or modify course content in an interactive online editor. The actual course content, formally called a topic, is stored inside a git repository on the filesystem. However, the platform also maintains a record of topic metadata in the relational database. This metadata includes title and description, but also permissions and contributors on the course content.

From a technical perspective, the user has three distinct paths of action for creation and

co-creation on course content. Since the permission model in the platform is not relevant to this research, we will not go into detail on it. First, the user can directly modify the course content if the user has permission to perform this action. The second option is creating annotations on the topic. This allows the user to attach private or public notes to specific content on the topic. Annotations are stored in the relational database, in the `Annotation` entity. The entity contains a logical pointer to the annotated content.

The final possibility to integrate user content into a topic is by adding comments. In contrast to annotations, comments have a typical structure. They can take the form of questions, notes, suggestions, and can also be nested - which allows simple interaction and conversation between multiple users, and effectively enables dialogue between students and teachers.

The intention of this thesis is to use the NoSQL data storage as storage mechanism for the Recent Activity feed. This entails that the authoritative information will not be stored in that data store, but rather be extracted from the relational database whenever an activity event is generated. Accordingly, some information may be omitted from this data store, while other information is copied.

Consider the following domain description structured as activity events in the Recent Activity feed. These events are items that a user may typically encounter in the web application as part of the feed.

" **John** created **Topic A**. "

" **Jane** commented on **Topic B**: *This is not a good example. Try and find a better one.* "

" **John** commented on **Jane's comment** on Topic B: *I agree.* "

" **Jane** annotated **Topic A**. "

" **Bob** updated **Topic B**. "

" **Bob** reacted to **John's comment** on Topic B. "

From this description we can already derive some requirements to take into account when designing the data models. Every event has a structure characterised by three aspects. First, there is a user at the base of the action, and in the descriptions this is the subject of every sentence. Second, the subject performs an action, and the actions are limited to a certain subset as determined by the developer. Third, the user operates on an object, which is usually but not always a topic.

In the activity events, the underlined text fragments represent hyperlinks in the web application to the relevant entities. A hyperlink for a user may link to the profile page of the user, or the contributions of the respective user on the relevant topic. Similarly, a hyperlink for a topic may link to a page that presents an overview of the topic, or directly to course content inside the topic. The actual destination is up to the developers of the platform, and is not directly relevant for this research. However, the existence of these hyperlinks entails

that every aspect previously described has to consist of at least one attribute that is used in the hyperlinks – most likely this will be the unique identifier of the entity in question.

We present the following conventions and rules to be followed in all NoSQL data models.

- The user of an activity is called the *subject*
- The action of an activity is called the *predicate*
- The predicate can be one of the following values:
  `created`, `updated`, `renamed`, `commented_on`, `annotated`, `reacted_to`
- The object to which an action refers, is called the *item*
- The item can reference topics and comments
- No additional attribute to facilitate hyperlinks will be included

Furthermore, since the data in the NoSQL data store is generated in function of the business-critical data in the platform, it is expected to be written to the database only once, and read many times. This enables us to design data models where read performance is prioritized over write performance. It is also important to note that the data is always queried in a reverse chronological way, since the Recent Activity feed displays the most recent events first.

Using this logical description of the domain, we can start to derive physical data models for the selected NoSQL data stores.

## 6.2 Physical data model

### 6.2.1 Language bindings

Table 5.1 lists for every NoSQL data store the programming languages for which language bindings are available. Some of these are developed, maintained and officially supported by the database vendor, while others blossomed forth from a community effort. For developing the physical data model, we have selected the most popular language bindings for Ruby and Ruby on Rails' ActiveModel according to RubyGems (RubyGems, 2003). If there is an official library available, it is preferred over a community library, with a view on the maintainability of the application.

For MongoDB, an officially supported language binding is available for plain Ruby and Ruby on Rails. The latter library, called Mongoid, integrates MongoDB directly into the Rails ecosystem and was chosen as a viable candidate for developing the physical data model and queries (MongoDB Inc., 2009c).

CouchDB on the contrary, does not have an officially supported Ruby binding. The most popular gem on RubyGems is CouchRest and its ActiveModel equivalent, CouchRest Model. However, as of the time of writing, CouchRest Model is not well maintained, with only five beta releases and no stable releases during the past three years. Since this library provides all the necessary integrations to develop the CouchDB data model and queries, it

was chosen as framework in which to implement these.

Finally, Neo4j does not have an officially supported library, however the database vendor recommends using a community supported alternative called Neo4j.rb. This library integrates the Neo4j graph database with the Ruby on Rails stack and was subsequently used as a platform to develop the graph data model and queries.

### 6.2.2 Document-oriented data model

The fundamental building block of a document data store is a document. The document data model provides two distinct approaches to link between different documents. Each option has its own advantages and disadvantages, and the proposed data models attempt to use the most efficient option for the use case, despite making some trade-offs.

1. **Embedded collections** (denormalized data). Embedding of data stores information in a single document. This technique is commonly used when the entity *contains* the embedded entity: for example storing contact details of a user. Another use for this approach is storing one-to-many relationships, where the child documents are always queried within the context of the parent document.
2. **References** (normalized data): Storing a reference to another document, similar to storing keys to other tables in the relational model.

Embedded collections provide better performance, since the embedded document is included in the parent document and the database management system does not have to execute an additional query. The disadvantage of embedding is that data may be duplicated, if an embedded document is included in multiple parent documents. Using referenced documents yields the exact opposite effects: slower performance due to additional queries, yet less data duplication in case of a multiply referenced document.

Since the data model has to be optimized for read performance, we will try to use embedded documents wherever possible.

From the domain description of the use case, we can identify one main entity which may be stored in its own collection: `Event`. This is the entrypoint of the Recent Activity feed, and the event document will embed or reference all other entities. The first relation that can be identified is the subject relation: every event has exactly one subject. However, we can infer that `Event` is not the only entity that has a link to `Subject`. Every comment also references `Subject`. Since the only information included in the `Subject` entity is a name, we have chosen to embed `Subject` in the parent documents. The data duplication of a single attribute is negligible as opposed tot the performance penalty encountered when using a referenced document in this case.

The same train of thought can be applied to `Topic`: both `Event` and `Comment` reference the entity, yet it only contains one attribute. Subsequently `Topic` will be used only as an embedded document as well.

One downside of this approach is that when a user changes the name or title of a subject or

topic respectively, the existing data in the NoSQL database does not get updated and the Recent Activity feed may display outdated information.

The way a comment gets stored in the document data store is very particular. The storage of both a comment and an event referencing the comment in different collections is redundant, since the database will never be queried from the perspective of a comment. Considering this, every top-level comment – comments made on a topic – is stored as a `text` attribute of an event. This way the `item` of that event still refers to the topic itself. Events for child comments – comments made on another comment – are structured differently. In this case, the `item` refers to the parent comment, but does not include its text. This flexible approach allows us to store the information efficiently.

This leads us to the following physical document data models, implemented in the Ruby on Rails application.

#### MongoDB implementation

The physical data model for MongoDB is implemented using the Mongoid library, which provides Ruby and Ruby on Rails bindings to the data store. Mongoid is officially supported by Mongo, Inc.

The models developed during this research are available in appendix B.

#### CouchDB implementation

An attempt was made to provide a physical implementation of the proposed document data model using the CouchRest Model library (Anderson et al., 2011). CouchRest Model provides Ruby on Rails integrations and is built on the CouchRest Ruby library.

During the development of the data models in question, many roadblocks were encountered that impeded development or even made it impossible to continue. The library does not support object inheritance, and relationship polymorphism for instance.

The degree in which CouchDB is compatible with Ruby on Rails was deemed not satisfactory, and as such CouchDB was dropped for the physical implementation of the queries.

The models developed during this research are available in appendix B.

### 6.2.3 Graph-oriented data model

Using the examples of activity events in section 6.1, several entities can be identified. These entities are used to model the *nodes*, *labels* and *edges* in the graph data stores

The first step is to extract the nodes from the description. In the domain, there are four main entities.

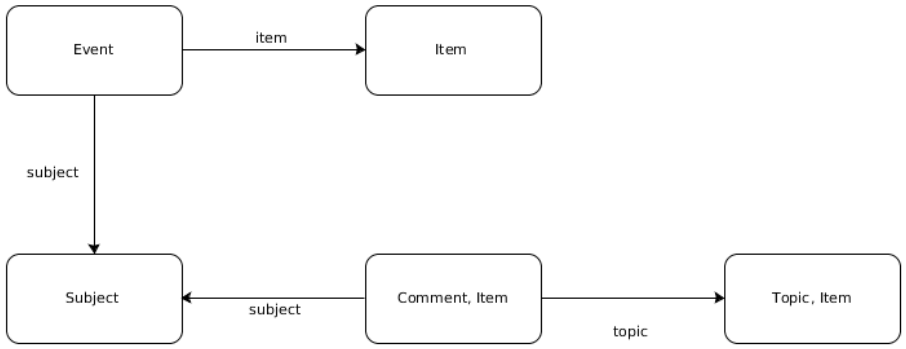


Figure 6.1: Graph data model

- Event
- Subject
- Topic
- Comment

Similarly to the document data model, `Event` represents an entry in the Recent Activity feed.

Neo4j data modeling also supports labels, a graph construct that groups nodes into sets. A set contains all nodes that are labeled with the same label. A node can have any number of labels.

In the use case, there is one important opportunity to make use of node labels. The `item` relation of `Event` can reference multiple other entities, in this case `Topic` and `Comment`. Using the label `Item` on top of the `Topic` and `Comment` labels allows room for future expansion to other entity types.

A number of relationships can be identified in the domain description.

- `Event` has one `Subject`
- `Event` has one `Item`
- `Comment` has one `Subject`
- `Comment` has one `Topic`

These relationships are modeled in the graph data store as edges.

This leads to the graph data model in figure 6.1.

### Neo4j implementation

The physical data model for Neo4j is implemented using the community-supported Neo4j.rb library, which provides Ruby and Ruby on Rails bindings to the data store (Underwood, 2010).

The models developed during this research are available in appendix B.

## 6.3 Reference queries

In order to perform an empricial analysis on the selected data stores and the proposed physical data models, we present five reference queries in this chapter. These reference queries will reflect the method of querying that would be the most common in the physical platform implementation, and mirrors the way data is queried from a database perspective. All reference queries will be implemented using the available language bindings, however the generated implementation-specific query will also be presented.

Since the data of the use case is aimed at a write-once, read-many character, the majority of queries will not touch the data itself, but rather only read it. Four read-only queries are included in the following sections, and one query that will insert new data into the data store.

### 6.3.1 Querying

#### Query 1

" Select N most recent events, ordered reverse chronologically "

This query, as most simple reference query presented, is an example of a query that can be used on the homepage of the platform. When a user opens the web application, a reverse chronologically ordered list of events is presented. This view allows for a quick overview of the activity in the platform, and since the user is not signed in yet, it is not tailored. This also means that everyone who visits the platform without signing in will receive the same events in their Recent Activity feed.

#### MongoDB

```
MongoDB::Event
  .all
  .order_by(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 1: MongoDB query 1

#### Neo4j

```
Neo4j::Event
  .all
  .order(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 2: Neo4j query 1

```
MATCH (result_neo4jevent:`Event`)
  RETURN result_neo4jevent
  ORDER BY result_neo4jevent.created_at DESC
  LIMIT {limit_1} | {:limit_1=>count}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`by`]->(next:`Subject`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>result_neo4jevent.id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`on`]->(next:`Item`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>result_neo4jevent.id}
```

Listing 3: Neo4j query 1 (CYPHER)

The Neo4j ORM call get converted into three distinct queries: one to get the `Event` node and two queries to get the related nodes `Subject` and `Item`.

### Query 2

" Select N most recent events, where the event is related to a topic in a list of given topics, ordered reverse chronologically "

A user has to ability to subscribe to topics, which means that the Recent Activity feed may be tailored to the user. Once the user logs in to the platform, the Recent Activity feed can be presented in a more attractive way. The events in the feed will then consist of only events related to topics the user has subscribed to (which also includes the topics where the user is author or contributor).

### MongoDB

```
# List of subscribed topic identifiers
topic_ids = [...]

MongoDB::Event
  .in('item._id' => topic_ds)
  .order_by(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 4: MongoDB query 2

### Neo4j

```
# List of subscripted topic identifiers
topic_ids = [...]

Neo4j::Topic
  .where(:id => topic_ids)
  .events
  .order_by(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 5: Neo4j query 2

```
MATCH (node2:`Topic`:`Item`)
  WHERE (node2.uuid IN {node2_uuid})
  MATCH (node2)<-[rel1:`on`]-(result_events:`Event`)
  RETURN result_events
  ORDER BY result_events.created_at DESC
  LIMIT {limit_1} | {:limit_1=>1, :node2_uuid=>topic_ids}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`by`]->(next:`Subject`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>node2.id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`on`]->(next:`Item`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>node2.id}
```

Listing 6: Neo4j query 2 (CYPHER)

### Query 3

“ Select N most recent events, where the event is related to a given topic, ordered reverse chronologically ”

Every topic also has an overview page, which mainly contains metadata such as description, author, contributors and other information not directly related to the course content. Next to the metadata, a custom Recent Activity feed is also included on the page. This feed only contains events related to the topic the user is currently viewing, and effectively presents a timeline of changes and discussions.

### MongoDB

```
# Topic identifier
topic_id = ...

MongoDB::Event
  .where('item._id' => topic_id)
  .order_by(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 7: MongoDB query 3

### Neo4j

```
# Topic identifier
topic_id = ...

Neo4j::Topic
  .find(topic_id)
  .events
  .order(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 8: Neo4j query 3

```
MATCH (neo4j_topic)
  WHERE (ID(neo4j_topic) = {ID_neo4j_topic})
  MATCH (neo4j_topic)<-[rel1:`on`]-(result_events:`Event`)
  RETURN result_events
  ORDER BY result_events.created_at DESC
  LIMIT {limit_1} | {:limit_1=>1, :ID_neo4j_topic=>topic_id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`by`]->(next:`Subject`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>neo4j_topic.id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`on`]->(next:`Item`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>neo4j_topic.id}
```

Listing 9: Neo4j query 3 (CYPHER)

### Query 4

“ Select N most recent events, where the event is related to a given user, ordered reverse chronologically ”

Similarly to topics, a profile page also includes a timeline of the user’s activities: additions, deletions, comments and annotations that were recently made by that user.

#### MongoDB

```
# Subject identifier
subject_id = ...

MongoDB::Event
  .where('subject._id' => subject_id)
  .order_by(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 10: MongoDB query 4

#### Neo4j

```
# Subject identifier
subject_id = ...

Neo4j::Subject
  .find(subject_id)
  .events
  .order(:created_at => :desc)
  .limit(count)
  .each(&:to_s)
```

Listing 11: Neo4j query 4

```
MATCH (n:`Subject`)
  WHERE (n.uuid = {n_uuid})
  RETURN n
  ORDER BY n.uuid
  LIMIT {limit_1} | {:n_uuid=>subject_id, :limit_1=>1}

MATCH (neo4j_subject)
  WHERE (ID(neo4j_subject) = {ID_neo4j_subject})
  MATCH (neo4j_subject)<-[rel1:`by`]-(result_events:`Event`)
  RETURN result_events
  ORDER BY result_events.created_at DESC
  LIMIT {limit_1} | {:limit_1=>1, :ID_neo4j_subject=>subject_id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`by`]->(next:`Subject`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>result_events.id}

MATCH (previous:`Event`)
  WHERE (ID(previous) = {ID_previous})
  OPTIONAL MATCH (previous)-[rel1:`on`]->(next:`Item`)
  RETURN
    ID(previous),
    collect(next) | {:ID_previous=>result_events.id}
```

Listing 12: Neo4j query 4 (CYPHER)

### 6.3.2 Insertion

#### Query 5

“ Insert N given :created or :updated events ”

Finally, since read requests will most likely outnumber write requests with several magnitudes in the studied use case, only one query where data is inserted is presented. The query creates a single event in the data store.

### MongoDB

```
# Subject identifier
subject_id = ...

# Item identifier
item_id = ...

MongoDB::Event.create! :subject => subject_id,
                       :item => item_id,
                       :predicate => :updated
```

Listing 13: MongoDB query 5

### Neo4j

```
# Subject identifier
subject_id = ...

# Item identifier
item_id = ...

Neo4j::Event .create! :subject => subject_id,
                      :item => topic_id,
                      :predicate => :updated
```

Listing 14: Neo4j query 5

```
MATCH (n:`Subject`)
  WHERE (n.uuid = {n_uuid})
  RETURN n
  ORDER BY n.uuid
  LIMIT {limit_1} | {:n_uuid=>subject_id, :limit_1=>1}

MATCH (n:`Item`)
  WHERE (n.uuid = {n_uuid})
  RETURN n
  ORDER BY n.uuid
  LIMIT {limit_1} | {:n_uuid=>item_id, :limit_1=>1}

CREATE (n:`Event`)
  SET n = {props}
  RETURN n | {:props=>{:uuid=>event_id, :created_at=>1526737892, :predicate=>1}}

MATCH (n:`Subject`)
  WHERE (n.uuid = {n_uuid})
  RETURN n
  LIMIT {limit_1} | {:n_uuid=>subject_id, :limit_1=>1}

MATCH
    (from_node),
    (to_node)
  WHERE
    (ID(from_node) = {ID_from_node}) AND
    (ID(to_node) = {ID_to_node})
  CREATE (from_node)-[rel:`by` {rel_create_props}]->(to_node)
    | {:ID_from_node=>event_id, :ID_to_node=>subject_id, :rel_create_props=>{}}

MATCH (n:`Item`)
  WHERE (n.uuid = {n_uuid})
  RETURN n
  LIMIT {limit_1} | {:n_uuid=>item_id, :limit_1=>1}

MATCH
    (from_node),
    (to_node)
  WHERE
    (ID(from_node) = {ID_from_node}) AND
    (ID(to_node) = {ID_to_node})
  CREATE (from_node)-[rel:`on` {rel_create_props}]->(to_node)
    | {:ID_from_node=>event_id, :ID_to_node=>item_id, :rel_create_props=>{}}
```

Listing 15: Neo4j query 5 (CYPHER)

## 6.4 Conclusion

In this chapter we presented an introduction to the domain, and provided some examples of events in the Recent Activity feed. Furthermore, we analyzed this domain description, and derived a logical data model for both document- and graph-oriented data stores. Next, we proposed an implementation of this logical data model for one document-oriented, and one graph-oriented data store in the Ruby language bindings available for the respective database management systems. Finally, we presented five reference queries for the three data stores, and included both a language binding-specific DSL and a physical query implementation for each.

The CouchDB implementation was dropped due to the most popular Ruby language binding available not being up to date and lacking many essential features.

# 7. Empirical study

In this chapter the previously proposed physical data models for MongoDB and Neo4j are implemented and put to the test. A Ruby on Rails application was developed that uses Mongoid and Neo4j.rb libraries to provide connectivity to the data stores. Custom benchmarking scripts were implemented using the RSpec library (Chelimsky et al., 2005) and the `Benchmark` class built into the Ruby implementation.

## 7.1 Previous work

In current literature there are already some studies present that compare NoSQL data stores based on a performance review. Abramova et al. (2014) compare the Cassandra, HBase, MongoDB, OrientDB and Redis data stores. The authors are using the Yahoo! Cloud Serving Benchmark (Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010), which presents a framework to facilitate performance comparisons for cloud-based systems by providing a core set of benchmarks. The paper concludes that Redis, as in-memory database, provides the best performance in query processing. Redis is optimized for *get* and *put* operations due to in-memory data mapping. On the other hand, Cassandra and HBase are optimized for update operations.

Finally, MongoDB was found to be the data store with the slowest execution times, having an overall performance that was more than 58 times lower in comparison with Redis. This proves that in-memory mapping of data results in a very performant query processing system.

Schmid et al. (2015) develop a performance comparison aimed at applications using geo-functionalities present in the database management system. The authors conclude that

requests purely on attributes NoSQL data stores are superior over relational data stores. For requests using geo-functions the NoSQL data stores also perform constant for increasing dataset sizes. For smaller datasets with a more interlinked architecture, the relational data stores perform predictably better.

Barahmand et al. (2015) quantify the horizontal and vertical scalability of MongoDB and HBase in the context of a social benchmarking framework named BG (Barahmand & Ghandeharizadeh, 2013). This benchmarking framework models a social graph in the data store and performs simple operations reading and writing small amounts of data with in a social interaction context. The researchers found that both data stores scale superlinearly, limited by the complete utilization of certain nodes in the cluster.

The experimental comparison by Kolomičenko, Svoboda, and Mlýnková (2013) concludes that Neo4j is the most performant product under the compared graph data stores, especially in graph traversal queries. The authors also indicate that MongoDB performs well in queries that are not or lightly graph related.

During the research of this thesis, the decision not to use BG or the Yahoo! Cloud Serving Benchmark was made. First, there is already literature in the field concerning these benchmarks and the data stores that were selected for comparison in this thesis. These sources provide additional input when formulating an answer on the research questions. However, developing custom benchmarks adjusted to the workload and environment the data schema is intended to be used in, present a more realistic view of real-word performance. This is coupled with the fact that this research delivers a Ruby implementation ready to be integrated into the existing platform.

## 7.2 Experimental setup

In order to keep the results of the tests consistent, the following rules are applied:

- Query caching is disabled in Mongoid. Neo4j.rb does not have an equivalent feature.
- Connection pooling is disabled
- Clustering is disabled as horizontal scalability performance is not within the scope of this research
- No additional performance tweaks were applied on the database management systems

Mongo Wire Protocol and Bolt Protocol were chosen as native protocols for MongoDB and Neo4j respectively.

All tests were performed on a single machine with the following specifications.

- Ruby 2.5.0, operating under Arch Linux
- Intel Core i7-3840QM (4 cores, 8 threads)
- Hyperthreading and Intel Turbo Boost enabled
- 32GB DDR4 RAM

- 180GB SATA-III SSD

The data store versions that were tested are the following versions.

- MongoDB 3.6.3
- Neo4j Community Edition 3.0.6

## 7.3 Procedure

The following procedure was followed throughout the performance benchmarks.

First, the database was filled with random testing data. The script included in appendix B was developed in order to create random data and insert this into each of the data stores. The variable `FACTOR` in the script is a multiplication factor that directly influences the amount of data generated.

Next, every test was ran sequentially for the data stores, and the timing results were written to separate files. During the tests all database management systems were running in the background.

Since the execution time of a single iteration of a query is negligable, every query was executed a number of times to negate the effect of external factors, such as operating system scheduling and I/O wait times. The effects of varying the iteration count are discussed in section 7.4.2. Having multiple iteration runs allows analyzing the vertical scalability of the query in limited fashion as well. As indicated in the informal descriptions of the reference queries, the query itself is also dependent on a variable $N$ which signifies the event count that is to be retrieved from the database. The effects of varying query sizes were analyzed and discussed in section 7.4.2. A sane default for query size in a concrete implementation could be 100, meaning that 100 events are retrieved every time the user loads the Recent Activity page.

In summary, there are three variables that can be modified in the performance tests:

- **Data multiplication factor**: total size of the dataset in the database
- **Iteration count**: number of sequentially ran iterations of the query
- **Query size**: number of retrieved events from the database

## 7.4 Results

Small-scale, informal tests determined that the Neo4j implementation is many times slower than the MongoDB implementation. Hence, the the benchmarks are ran independently for MongoDB and Neo4j, using an iteration count that is magnitudes smaller for Neo4j, yet yielding roughly the same execution times.

The execution times represent the real wall-clock time elapsed for every query. It is

| Multiplication factor | Dataset size |
|---|---|
| 500 | 7 250 events |
| 1000 | 14 500 events |
| 2500 | 36 250 events |
| 5000 | 72 500 events |

Table 7.1: Multiplication factor and dataset size

measured from the moment the query gets dispatched to the ORM, and hence includes the time to traverse the entire software stack, including instantiation of the database objects in Ruby. This overhead is intended to be included in the measurements, since the total time a user has to wait for a database query is influenced both by the software stack and the database management system.

### 7.4.1 Dataset size

By varying the multiplication factor in the seed data generator the size of dataset can be controlled. The effects of dataset size on query performance were not tested and discussed in this paper. Since the system will most likely be scaled horizontally before dataset will reach a sufficiently large enough level to be measurable, this variable was not benchmarked. Horizontal scalability is not within the scope of this research.

All queries below were ran on a dataset with multiplication factor 5000, which implies that 72 500 events are stored in the database.

### 7.4.2 Query size

The amount of events retrieved from the database is another aspect that could possibly influence the query performance heavily. In this section different values for the query size are compared, in order to find any trends

MongoDB repeats the following 10 000 times, while Neo4j only uses an iteration count of 10. All queries operate on a dataset with multiplication factor of 5000, yielding 72 500 events in the database.

Queries 1, 2 and 3 in section 6.3.1 were used as queries in this test. Query 4 is very similar to query 3 in structure of the queried data itself, so it was omitted. Query 5 is a query that tests insertion of data, and it was omitted as well due to the fact that query size is not relevant for insertion queries.


Figure 7.1: MongoDB query size

The horizontal axis in figure 7.1 represents query size in the logarithmic scale. The vertical axis represents the execution time of 10 000 iterations of the query in seconds.

All measured execution times fall roughly within 7 and 9 seconds, which means that the query execution time is constant for a varying query size in MongoDB

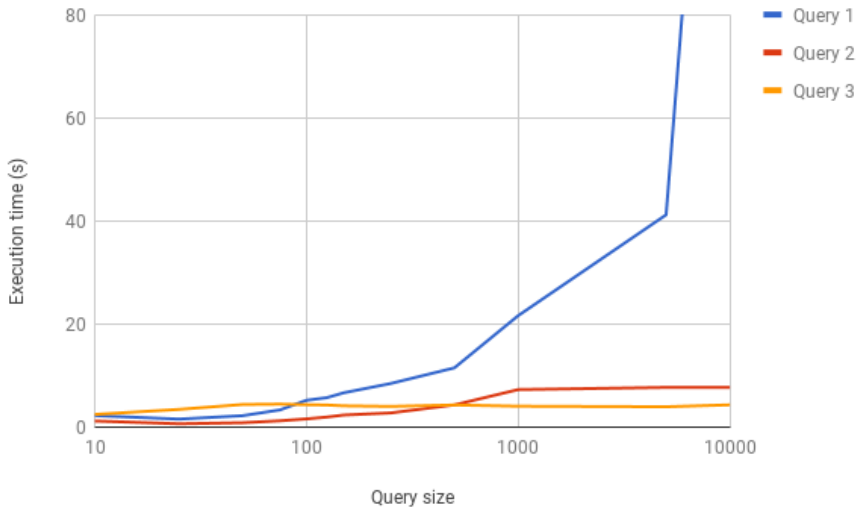


Figure 7.2: Neo4j query size

Neo4j however, renders a completely different result. Figure 7.2 plots out the execution time for the different queries running on the Neo4j graph database management system. The first difference with MongoDB is already very apparent in iteration size: MongoDB can handle roughly 1000 times as many requests in the same execution time. This is due to

the fact that for every entity retrieved from the database, MongoDB only has to execute one request and retrieve one document, while Neo4j's inherent linked structure means that at least three entities have to be retrieved (event, subject and item).

Query 2 and query 3 remain constant, similar to their MongoDB equivalents. Query 1 however, rises exponentially with query size. This is not an expected result since a database index exists for the `created_at` field on which the query orders. However, for small, realistic values of query size this should not pose a problem.

### 7.4.3 Iteration count

Varying the iteration size of the queries simulates sequental requests to the data store, handled by the server. Using these tests we attempted to determine how the queries and the data stores would behave when handling an increased workload.

The multiplication factor in these tests was set to 5000, similar to the previous tests. Query size for all queries is 100.

Analogous to the previous test, query 4 was not included in the benchmark.

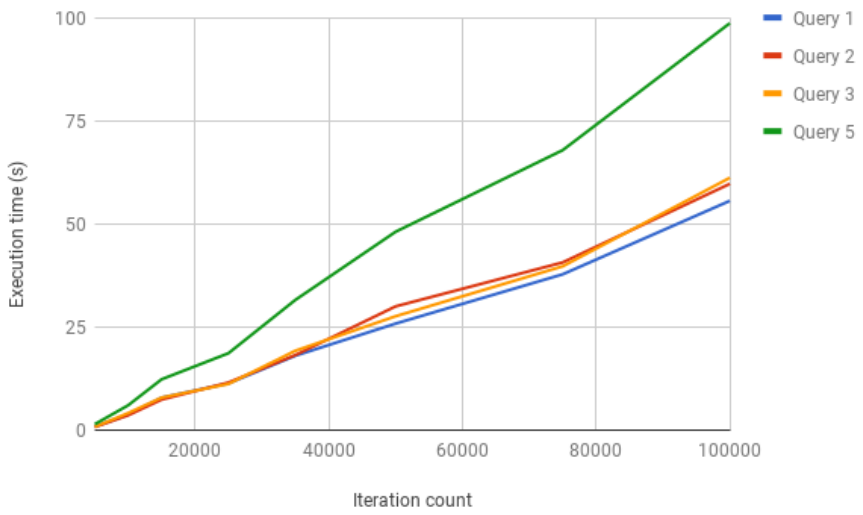


Figure 7.3: MongoDB iteration count

All MongoDB query execution times rise linearly in function of the iteration count. The first query gets slower a little bit faster than the other queries. This is due to the fact that the query operates on the entire dataset, whereas the other queries only consider part of the dataset. The effect of sorting through the entire dataset instead of a slice of it is not measurable within a single iteration, but it becomes visible when the iteration size is increased.

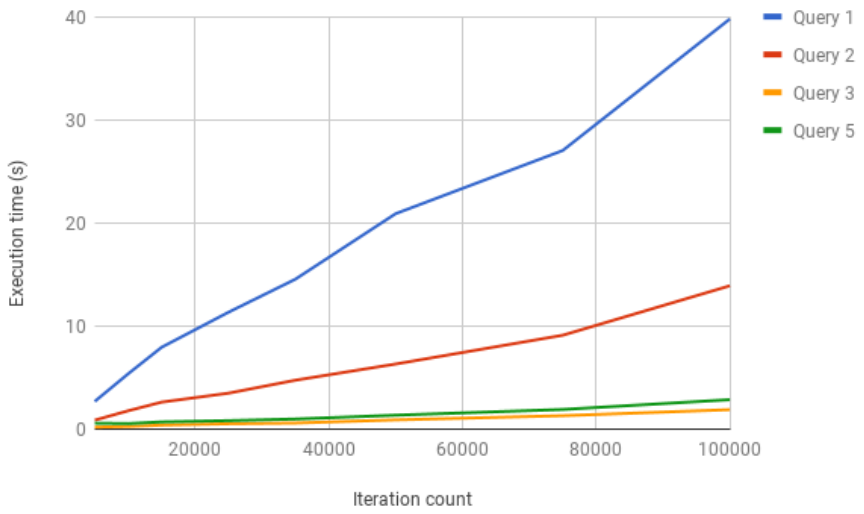


Figure 7.4: Neo4j iteration count

The Neo4j query execution times exhibit a behavious similar to the MongoDB query execution times. The first query is a lot slower than the others, and this effect is only amplified in the graph database. While query 2 also decreases in speed in function of iteration count, it is not as dramatic as the first query.

## 7.5 Conclusion

We can conclude that MongoDB is a far superior solution for this specific use case. However, the data models, queries and database settings were not optimized at all, leaving the defaults in place.

# 8. Opportunities

Since the field of NoSQL is comprised of a lot of different products, varying in data model and features, choosing the right data store for a use case is a difficult and complex choice. The analysis and comparison of the NoSQL data stores in this research has revealed certain pain points and opportunities for future research.

First, a common terminology needs to be established for NoSQL data stores, at least for data stores within the same NoSQL category that adhere to the same data model. Comparing data stores using different vocabulary for the same concepts is confusing and impedes comparison and analysis of these products. Establishing a common terminology in the NoSQL field will not only allow to make a more informed decision and doing this more efficient, but also facilitates the switch between different NoSQL products.

Secondly, the difficulty and time penalties imposed by the proposed implementation of the data models and reference queries in the three compared NoSQL data stores, proved that there certainly is an opportunity for a common querying language between NoSQL products, at least for data stores that have the same data model. For example, the difference in interface between MongoDB's JavaScript API and CouchDB's REST API – while still having the same data model – does not allow for an easy switch between the two data stores, effectively requiring a complete rewrite and -engineering of the data storage layer in the application. Standardizing the querying of NoSQL data stores – at least per category – would likely help the adoption of NoSQL storage systems and lower developer learning curve.

Some experimental solutions already exist to aggregate querying on certain data stores. Hive provides an interface that uses SQL-like semantics to allow querying multiple data stores, however this project is in practice limited to a small number of backing stores, most

importantly HBase and Cassandra (Apache Software Foundation, 2017). Another proposed solution is UnQL, a Unified Querying Language for NoSQL data stores (Buneman, Fernandez, & Suciu, 2000). This collaboration between Couchbase and SQLite aimed to implement a unified API covering all NoSQL data stores, from document-oriented to graph data stores. However, seeing the massive scope of this project, it seems to have suffered from a severe case of hybris and is no longer being developed (Weinberger, 2012).

There exists a similar discrepancy in data modeling for NoSQL data storage solutions. Every product and more generally every NoSQL data store category imposes its own implementation-specific data modeling layer. Since reimplementation and adjustment of the data storage layer of the application is a far reaching and costly operation, designing an application's data model for an intermediate, abstract data model before approaching the practical implementation would be a solution for this problem. Bugiotti et al. suggests a novel abstract data model for NoSQL systems called NoAM (NoSQL Abstract Model), which exploits certain commonalities between different NoSQL data models, allowing developers to model information in an intermediate format that can be adapted to multiple storage systems.

Additionally, RDF triple stores were not considered in the scope of this research. However, similarly to key-value stores, usage of RDF semantics and the performance implications of data stores using simple data models may provide an interesting entrypoint for future research as an applied case study.

Finally, benchmarking NoSQL solutions is also a pain point encountered in this research. There is no standardized way to provide reliable benchmarks of the different NoSQL categories, in part due to the data models that differ conceptually. Any performance comparisons have to be done either at an atomic entity level, or a global use-case level as is presented in this thesis in order to obtain results that are reliable and not tainted by any modeling-specific conditions. Furthermore, plenty of important factors were not considered in this research. The performance benchmarks do not take into account the horizontal scalability of NoSQL solutions, and use the default configuration for each product. Finetuning the database management system parameters would allow performance gains without any additional logical design or implementation. Finally, since this thesis only represents a minor venture into NoSQL data modeling, the presented data models can also be improved upon.

# 9. Conclusion

The intended aim of NoSQL data stores is to store and process massive amounts of data in a distributed fashion. However, there exist multiple data models which store data in a different way, which can be more or less performant to the applied use case. NoSQL data stores are not “one size fits all”.

In this paper all proposed research questions were satisfactorily answered. First, a literature review offered insight in the NoSQL products currently available on the open source and commercial market. The different data models were discussed by advantages and disadvantages.

Second, a conceptual and technical definition of the Open Webslides data model was designed and analyzed. This proved to be a valuable resource to proceed with the design and development of the physical data models and corresponding implementation in the MongoDB and Neo4j graph data stores. The second research question was answered by these presented data models, along with the five reference queries that were drawn up and represent typical usage in the studied use case.

Finally, as most efficient data store MongoDB was selected as NoSQL candidate. Subsequently, the recommendation for the Open Webslides development team is to use the MongoDB as a polyglot persistent NoSQL database to store the derived data for the Recent Activity feed.

In contrast to the expected results, using the Neo4j data store turned out to be magnitudes slower in performing the reference queries on the developed data model. Instead, MongoDB demonstrated that the document data model is far superior over the graph data model in this specific use case. The fact that the data from which the Recent Activity feed is derived is highly interlinked and easily structured as a graph, counteracts the expectations

of a highly linked data structure to be efficient.

It was found that MongoDB has excellent support for Ruby and Ruby on Rails using the Mongoid ORM framework. For all these reasons MongoDB is the clear winner as performant NoSQL data store for the Open Webslides' Recent Activity feed.

This research presents a case study of NoSQL data store selection for a specific use case. The paper may offer additional opportunities for research as described in chapter 8, and can be used as a guideline to pick the right data store for other, similar use cases.

Finally, the physical implementation of the data schema proposed in this thesis can be implemented in the Open Webslides platform to integrate a highly scalable, performant polyglot infrastructure to accommodate the Recent Activity feed.

# A. Research proposal

The subject of this thesis is based upon a research proposal that was graded by the promotor in advance. This proposal has been included as an attachment.

## A.1 Introduction

The Open Webslides (2017b) project provides a user-friendly platform to collaborate on webslides - slides made with modern web technologies such as HTML, CSS and JavaScript. One of the core features this application provides is *co-creation*. The co-creation aspect manifests itself in several forms within the application; annotations on slides and a change suggesting system resembling GitHub's pull request feature are the main mechanisms. Because of the inherent social nature of co-creation, a basic notifications feed was also implemented. This feed is tailored to the user, and reflects the most recent changes, additions and comments relevant to the slide decks the user is interested in.

However, at the moment the functionality implemented in the system contains only the bare necessities. The module will be expanded in the future, and doing so requires a structural and conceptual rethinking of how the notifications are generated, stored and queried. The size of the dataset is also expected to grow linearly with user activity, therefore scalability is a requirement as well.

This paper has two research questions.

1. What frameworks and software packages currently exist in the industry to store structured non-relational graph or document data?
2. How is the social graph provided by the Open Webslides' notification feed structured

and how is this data consumed?

Answering these questions is paramount for the final section of the paper, which describes a data storage mechanism that performs well given the functional requirements of the project's data flow.

## A.2 Use case

This thesis is a study of NoSQL data storage techniques applied to the Open Webslides (2017b) project. The online, interactive platform that the project provides includes a list of notifications in reverse chronological order, tailored to the user. This feature is called the *social feed*. It enumerates the most recent social activity on the platform. For example, if a user updates a slide deck, the user's friends would be able to find a change notification in their respective social feeds.

From the perspective of the application code that generates the social feed, the user, slide deck and notification should be considered separate entities. The notification itself has relations to the other entities: the author (the *subject*), the slide deck (the *object*), along with the *predicate* property that provides information on what operation was performed (for example creating or updating a slide deck).

This data model is also characterized by the *write-once read-many* nature of the information; once the notification has been generated, it does not need to be modified again. It will also only be queried in a very specific way: the application server will always attempt to retrieve the most recent notifications starting from the user entity. This principle is an important aspect to take into consideration in the choice of data storage mechanism.

## A.3 State-of-the-art

In current literature, studies such as Moniruzzaman and Hossain (2013), Nayak, Poriya, and Poojary (2013b) and Dayne Hammes and Mitchell (2014) have already analyzed the disparity between traditional relational database systems and NoSQL stores. However, the conceptual and technical difference between these data storage models will not be scrutinized any further, since this paper presents a data storage solution applied to the social notification feed of the Open Webslides (2017b) project.

There are already many existing free and commercial products for the storage of NoSQL data, such as Redis (Sanfilippo, 2009), CouchDB (Apache Software Foundation, 2005c), MongoDB (Inc., 2009) and Neo4j (Technology, 2007). Finding the right database model for this use case (section A.2) is one of the hurdles this paper intends to handle. Zhao (2015) describes the development of a messaging system for astrophysical transient event notifications. Part of this paper is a qualitative comparison between document-based NoSQL storage solutions fit for this particular use case. We expect this paper to provide

a solid base of reasoning in order to find a scalable and efficient solution for resolving similar computational challenges.

The goal of this paper is to provide a performant, scalable and maintainable data storage schema for the Open Webslides (2017b) platform, regarding the linked social graph that powers the *Social Feed* functionality present in the platform.

## A.4 Methodology

First, a range of industry-standard NoSQL database management systems such as MongoDB (Inc., 2009), HBase (Apache Software Foundation, 2005d) and Neo4j (Technology, 2007) will be qualitatively analyzed. Three of the five types of NoSQL database types (Nayak et al., 2013b) will be included in the study: column-oriented, document based and graph databases. Criteria for comparison include how the database management system concretely stores its data on disk, the query format and specific programming language bindings. Another important aspect is the distributed nature of many NoSQL databases. Using Brewer's conjecture (Gilbert & Lynch, 2002b, 2) – often called the CAP theorem – the existing types of data storage systems will be examined and summarized. There is also a practical factor present in the research; this includes the license of the project, its active maintainability and future prospects. Common types of NoSQL databases include key-value store, column-oriented, document store and graph databases (Nayak et al., 2013b). This paper will provide a short introduction to these types, before proceeding with the type that fits our use case the most.

Second, the data model specific to the Open Webslides project will be examined. We will start from the data model that is already implemented in the current iteration of the platform. At the time of writing, the existing base implementation of the social notification feed only contains two types of notifications. This paper will try to extrapolate this concept into a more generalized, abstract system in which developers can easily plug additional notification types. The physical properties of the data model will also be taken into account: the data will be written to the data storage only once, but read many times. It is also highly interlinked information, as a notification will always relate to one or more users as a subject, and a target object as well – most likely a slide deck or collection of slide decks. These links need to be maintained, and efficiently reconstructed when queried.

Finally, a sample dataset will be constructed using the aforementioned detailed analysis. Empirical testing will be conducted against multiple database management systems, and the results will be summarized and interpreted. Various information flows will be tested; however, the most important process remains efficiently querying the stored data.

Using the comparative study of storage engines, data model analysis and the empirical results an implementation plan will be constructed. This plan will serve as a recommendation for future development.

## A.5 Expected results and conclusions

The NoSQL ecosystem, unlike relational databases, is headed towards specialization, so different solutions are headed in different directions (Maroo, 2013). In this paper, we expect to find one type of NoSQL database that is a better fit for the Open Webslides use case, in clear contrast with the other types of storage engines. Due to the inherently highly interlinked nature of the stored data, we suspect a graph-based database management system to provide most advantages, and generally the most performant experience.

This expectation is amplified by the availability and good community support of Ruby bindings to the most popular graph database management systems.

Since the platform being discussed only caters to a small to medium user base, we do not expect the need to scale horizontally beyond one instance. However, the vertical scalability is still a topic for discussion, and we expect to determine the computational order of magnitude in order to efficiently query the given dataset during this study.

Finally, the implementation plan should describe a concrete roadmap, stretching over a development period with a baseline expectation of one to three months. Roll-out of this mechanism should also be included in this plan.

We also expect that this thesis will provide a good reference to a further stable, scalable and extendable implementation of the *social feed* feature in the Open Webslides (2017b) project as outlined in section A.2.

# B. Source code

## B.1 MongoDB

### Event

```
# frozen_string_literal: true

module MongoDB
  class Event
    include Mongoid::Document
    include Mongoid::Timestamps::Created
    extend Enumerize

    ##
    # Properties
    #
    field :predicate,
          :type => String

    enumerize :predicate,
              :in => %i[created updated renamed commented_on annotated reacted_to],
              :predicates => true

    field :text,
          :type => String

    ##
    # Relationships
    #
    embeds_one :subject,
               :class_name => 'MongoDB::Subject'
```

```
    embeds_one :item,
               :class_name => 'MongoDB::Item',
               :as => :event

    ##
    # Validations
    #
    validates :predicate,
              :presence => true

    validates :text,
              :presence => true,
              :if => :commented_on?

    validates :subject,
              :presence => true

    validates :item,
              :presence => true

    ##
    # Methods
    #
    def to_s
      if predicate == :commented_on
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}: #{text}"
      else
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}"
      end
    end
  end
end
```

## Comment

```
# frozen_string_literal: true

module MongoDB
  class Comment < Item
    ##
    # Properties
    #
    ##
    # Relationships
    #
    embeds_one :subject,
               :class_name => 'MongoDB::Subject'

    embeds_one :topic,
               :class_name => 'MongoDB::Topic'

    ##
    # Validations
```

```
    #
    validates :subject,
              :presence => true

    validates :topic,
              :presence => true

    ##
    # Methods
    #
    def to_s
      "#{subject}'s comment on #{topic}"
    end
  end
end
```

### Item

```
# frozen_string_literal: true

module MongoDB
  class Item
    include Mongoid::Document

    ##
    # Properties
    #
    field :_type

    ##
    # Relationships
    #
    embedded_in :event,
                :polymorphic => true,
                :class_name => 'MongoDB::Event'

    ##
    # Validations
    #
    ##
    # Methods
    #
  end
end
```

### Subject

```
# frozen_string_literal: true

module MongoDB
  class Subject
    include Mongoid::Document
```

```
    ##
    # Properties
    #
    field :first_name,
          :type => String

    ##
    # Relationships
    #
    embedded_in :event,
                :class_name => 'MongoDB::Event'

    ##
    # Validations
    #
    validates :first_name,
              :presence => true

    ##
    # Methods
    #
    def to_s
      first_name
    end
  end
end
```

## Topic

```
# frozen_string_literal: true

module MongoDB
  class Topic < Item
    ##
    # Properties
    #
    field :title,
          :type => String

    ##
    # Relationships
    #
    ##
    # Validations
    #
    validates :title,
              :presence => true

    ##
    # Methods
    #
    def to_s
      "'#{title}'"
    end
```

```
  end
end
```

## B.2 CouchDB

### Event

```
# frozen_string_literal: true

module CouchDB
  class Event < CouchRest::Model::Base
    extend Enumerize

    ##
    # Properties
    #
    property :predicate,
             String

    enumerize :predicate,
              :in => %i[created updated renamed commented_on annotated reacted_to]

    property :text,
             String

    property :created_at,
             Time

    ##
    # Relationships
    #
    property :subject,
             CouchDB::Subject

    property :item,
             CouchDB::Item

    ##
    # Validations
    #

    validates :predicate,
              :presence => true

    validates :text,
              :presence => true,
              :if => -> { predicate == :commented_on }

    validates :created_at,
              :presence => true

    validates :subject,
              :presence => true

    validates :item,
              :presence => true

    ##
```

```
    # Methods
    #
    def to_s
      if predicate == :commented_on
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}: #{text}"
      else
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}"
      end
    end

    ##
    # Views
    #
    design do
      ## Return events ordered by created_at
      view :by_created_at,
           :map => "function(doc) {
             if (doc['#{model_type_key}'] == 'CouchDB::Event') {
               emit(doc.created_at, doc);
             }
           }"

      ## Return events with topic item, ordered randomly
      view :random_with_topic,
           :map => "function(doc) {
             if (doc['#{model_type_key}'] == 'CouchDB::Event' && doc.item.item_type == 'topic') {
               emit(Math.random(), doc);
             }
           }"

      # Return events with topic item
      view :with_topic,
           :map => "function(doc) {
             if (doc['#{model_type_key}'] == 'CouchDB::Event' && doc.item.item_type == 'topic') {
               emit(doc.created_at, doc);
             }
           }"
    end
  end
end
```

## Item

```
# frozen_string_literal: true

module CouchDB
  class Item < CouchRest::Model::Base
    include CouchRest::Model::Embeddable
    extend Enumerize

    ##
    # Properties
    #
    property :item_type,
```

```
            String

    enumerize :item_type,
              :in => %i[topic comment]

    ## Subject properties
    property :title,
             Symbol

    ## Comment properties
    property :subject,
             CouchDB::Subject

    property :topic,
             CouchDB::Item

    ##
    # Relationships
    #
    ##
    # Validations
    #
    validates :item_type,
              :presence => true

    ## Subject properties
    validates :title,
              :presence => true,
              :if => -> { item_type == :topic }

    ## Comment properties
    validates :subject,
              :presence => true,
              :if => -> { item_type == :comment }

    validates :topic,
              :presence => true,
              :if => -> { item_type == :comment }

    ##
    # Methods
    #
    def to_s
      case item_type
      when 'topic'
        "'#{title}'"
      when 'comment'
        "#{subject}'s comment on #{topic}"
      else
        'item'
      end
    end
  end
end
```

### Subject

```ruby
# frozen_string_literal: true

module CouchDB
  class Subject < CouchRest::Model::Base
    include CouchRest::Model::Embeddable

    ##
    # Properties
    #
    property :first_name,
             String

    ##
    # Relationships
    #
    ##
    # Validations
    #
    validates :first_name,
              :presence => true

    ##
    # Methods
    #
    def to_s
      first_name
    end
  end
end
```

## B.3 Neo4j

### Event

```ruby
# frozen_string_literal: true

module Neo4j
  class Event
    include Neo4j::ActiveNode
    include Neo4j::Timestamps::Created

    ##
    # Properties
    #
    enum :predicate => %i[created updated renamed commented_on annotated reacted_to]

    property :text,
             :type => String

    ##
    # Relations
    #
    has_one :out,
            :subject,
            :type => :by,
            :model_class => 'Neo4j::Subject'

    has_one :out,
            :item,
            :type => :on,
            :model_class => 'Neo4j::Item'

    ##
    # Validations
    #
    validates :predicate,
              :presence => true

    validates :text,
              :presence => true,
              :if => :commented_on?

    validates :subject,
              :presence => true

    validates :item,
              :presence => true

    ##
    # Methods
    #
    def to_s
      if predicate == :commented_on
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}: #{text}"
      else
```

```
        "#{subject} #{predicate.to_s.humanize.downcase} #{item}"
      end
    end
  end
end
```

### Comment

```
# frozen_string_literal: true

module Neo4j
  class Comment < Item
    ##
    # Properties
    #
    ##
    # Relations
    #
    has_one :out,
            :subject,
            :type => :by,
            :model_class => 'Neo4j::Subject'

    has_one :out,
            :topic,
            :type => :on,
            :model_class => 'Neo4j::Topic'

    ##
    # Validations
    #
    validates :subject,
              :presence => true

    validates :topic,
              :presence => true

    ##
    # Methods
    #
    def to_s
      "#{subject}'s comment on #{topic}"
    end
  end
end
```

### Item

```
# frozen_string_literal: true

module Neo4j
  class Item
    include Neo4j::ActiveNode
```

```
    ##
    # Properties
    #
    ##
    # Relationships
    #
    has_many :in,
             :events,
             :type => :on,
             :model_class => 'Neo4j::Event'

    ##
    # Validations
    #
    ##
    # Methods
    #
  end
end
```

## Subject

```
# frozen_string_literal: true

module Neo4j
  class Subject
    include Neo4j::ActiveNode

    ##
    # Properties
    #
    property :first_name,
             :type => String

    ##
    # Relations
    #
    has_many :in,
             :events,
             :type => :by,
             :model_class => 'Neo4j::Event'

    ##
    # Validations
    #
    validates :first_name,
              :presence => true

    ##
    # Methods
    #
    def to_s
      first_name
```

```
    end
  end
end
```

### Topic

```
# frozen_string_literal: true

module Neo4j
  class Topic < Item
    ##
    # Properties
    #
    property :title,
             :type => String

    ##
    # Relations
    #
    ##
    # Validations
    #
    validates :title,
              :presence => true

    ##
    # Methods
    #
    def to_s
      "'#{title}'"
    end
  end
end
```

## B.4 Empirical study

### Seeds

```
# frozen_string_literal: true

## Variables
# Test data multiplication factor
FACTOR = 5000

##
# Global utility functions
#
def random(model)
  model.skip(rand(model.count)).first
end

# https://gist.github.com/0-I/3e0654509dd8057b539a
def weighted_rand(freq)
  freq.max_by { |_, weight| rand**(1.0 / weight) }.first
end

##
# Seeding
#

require_relative 'seeds/mongo_db'
require_relative 'seeds/neo4j'
```

### MongoDB

```
# frozen_string_literal: true

puts 'Seeding MongoDB database'

##
# Clear database
#
[
  MongoDB::Event,
].each(&:delete_all)

##
# Seed database
#

## Subjects (embedded)
puts 'Creating subjects'
subjects = (FACTOR / 5).times.map { FactoryBot.build :mongodb_subject }

## Topics (embedded)
puts 'Creating topics'
topics = FACTOR.times.map { FactoryBot.build :mongodb_topic }
```

```
## Events
puts 'Creating events'
topics.each do |topic|
  subject = subjects.sample

  MongoDB::Event.create! :subject => subject,
                         :item => topic,
                         :predicate => :created,
                         :created_at => Faker::Time.backward(365)

  3.times do
    MongoDB::Event.create! :subject => subject,
                           :item => topic,
                           :predicate => weighted_rand(:updated => 0.9, :renamed => 0.1),
                           :created_at => Faker::Time.backward(365)
  end
end

(FACTOR * 5).times do
  MongoDB::Event.create! :subject => subjects.sample,
                         :item => topics.sample,
                         :predicate => :annotated,
                         :created_at => Faker::Time.backward(365)
end

## Comments
puts 'Creating comments'
comments = (FACTOR * 3).times.map do
  FactoryBot.build :mongodb_comment,
                   :subject => subjects.sample,
                   :topic => topics.sample
end

items = (topics + comments)
(FACTOR * 5).times do
  MongoDB::Event.create! :subject => subjects.sample,
                         :item => items.sample,
                         :predicate => :commented_on,
                         :text => Faker::Lorem.words(20).join(' ').capitalize,
                         :created_at => Faker::Time.backward(365)
end

(FACTOR / 2).times do
  MongoDB::Event.create! :subject => subjects.sample,
                         :item => items.sample,
                         :predicate => :reacted_to,
                         :created_at => Faker::Time.backward(365)
end
```

### Neo4j

```
# frozen_string_literals: true

puts 'Seeding Neo4j database'
```

```
tx = Neo4j::ActiveBase.current_session.transaction

##
# Clear database
#
[
  Neo4j::Comment,
  Neo4j::Item,
  Neo4j::Subject,
  Neo4j::Topic,
].each(&:delete_all)

##
# Seed database
#

## Subjects
puts 'Creating subjects'
(FACTOR / 5).times { FactoryBot.create :neo4j_subject }
subjects = Neo4j::Subject.all.to_a

## Topics
puts 'Creating topics'
FACTOR.times { FactoryBot.create :neo4j_topic }
topics = Neo4j::Topic.all.to_a

## Events
puts 'Creating events'
Neo4j::Topic.all.each do |topic|
  subject = subjects.sample

  Neo4j::Event.create! :subject => subject,
                       :item => topic,
                       :predicate => :created,
                       :created_at => Faker::Time.backward(365)

  3.times do
    Neo4j::Event.create! :subject => subject,
                         :item => topic,
                         :predicate => weighted_rand(:updated => 0.9, :renamed => 0.1),
                         :created_at => Faker::Time.backward(365)
  end
end

(FACTOR * 5).times do
  Neo4j::Event.create! :subject => subjects.sample,
                       :item => topics.sample,
                       :predicate => :annotated,
                       :created_at => Faker::Time.backward(365)
end

## Comments
puts 'Creating comments'
(FACTOR * 3).times.map do
  FactoryBot.create :neo4j_comment,
                    :subject => subjects.sample,
```

```
                    :topic => topics.sample
end
comments = Neo4j::Comment.all.to_a

items = (topics + comments)
(FACTOR * 5).times do
  Neo4j::Event.create! :subject => subjects.sample,
                       :item => items.sample,
                       :predicate => :commented_on,
                       :text => Faker::Lorem.words(20).join(' ').capitalize,
                       :created_at => Faker::Time.backward(365)
end

(FACTOR / 2).times do
  Neo4j::Event.create! :subject => subjects.sample,
                       :item => items.sample,
                       :predicate => :reacted_to,
                       :created_at => Faker::Time.backward(365)
end

tx.commit
```

### Benchmark support

```
require 'benchmark'

# Number of iterations to perform
ITERATIONS = [
  1_000,
  10_000,
  100_000,
].freeze

# Count of queried events
COUNT = [
  100,
  1_000,
  10_000,
].freeze

##
# Execute block benchmark according to parameters
#
def benchmark
  ITERATIONS.each do |iteration|
    COUNT.each do |count|
      it "executes #{iteration} iterations with limit #{count}" do
        b = Benchmark.measure do
          iteration.times { yield count }
        end

        puts 'it = %8s, co = %4s, ti = %3ss' % [iteration, count, b.real.round(2)]
      end
    end
  end
end
```

# Bibliography


Abadi, D. J. [Daniel J.], Madden, S. R., & Hachem, N. (2008). Column-stores vs. Row-stores: How Different Are They Really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (pp. 967–980).

Abadi, D. J. [Daniel J], Boncz, P. A., & Harizopoulos, S. (2009). Column-oriented Database Systems. *Proceedings of the VLDB Endowment*, *2*(2), 1664–1665.

Abramova, V., Bernardino, J., & Furtado, P. (2014). Which NoSQL Database? A Performance Overview. *Open Journal of Databases*, *1*(2).

Anderson et al. (2011). CouchRest Model. Retrieved from https://github.com/couchrest/couchrest_model

Apache Software Foundation. (2005a). Consistent Hashing. Retrieved from http://guide.couchdb.org/draft/clustering.html#hashing

Apache Software Foundation. (2005b). CouchDB. Retrieved from https://couchdb.apache.org/

Apache Software Foundation. (2005c). CouchDB. Retrieved from https://couchdb.apache.org/

Apache Software Foundation. (2005d). Neo4j. Retrieved from https://hbase.apache.org/

Apache Software Foundation. (2017). Hive. Retrieved from https://hive.apache.org/

Atzeni, P., Bugiotti, F., Cabibbo, L., & Torlone, R. (2016). Data Modeling in the NoSQL World. *Computer Standards & Interfaces*.

Barahmand, S. & Ghandeharizadeh. (2013). *BG: A Benchmark to Evaluate Interactive Social Networking Actions*. University of Southern California.

Barahmand, S., Ghandeharizadeh, S., & Li, J. (2015). *On Scalability of Two NoSQL Data Stores for Processing Interactive Social Networking Actions*. University of Southern California.

Barrigas, H., Barrigas, D., Barata, M., Bernardino, J., & Furtado, P. (2015). Scalability of Facebook Architecture. In *New Contributions in Information Systems and Technologies* (pp. 763–772). Springer International Publishing.

Bernstein, P., Hadzilacos, V., & Goodman, N. (1987). Concurrency control and recovery in database systems.

Brewer, E. (2000). Towards Robust Distributed Systems.

Bugiotti, F., Cabibbo, L., Atzeni, P., & Torlone, R. (2014). Database Design for NoSQL Systems. In *International Conference on Conceptual Modeling* (pp. 223–231).

Buneman, P., Fernandez, M., & Suciu, D. (2000). UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, *9*(1), 76–110.

Cattell, R. (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Rec. 39*(4), 12–27.

Chelimsky et al. (2005). RSpec. Retrieved from http://rspec.info/

Co-created Courses through Open Source initiatives. (2018).

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, *13*(6), 377–387.

Codd, E. F. (1985). Is Your DBMS Really Relational? *Computerworld*, *19*(41), 1–2.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (pp. 143–154). ACM.

Cottenier, S., Verstraete, A., Verborgh, R., Brysbaert, M., De Loof, E., & Janssens, C. (2016). Aanvraag onderwijsinnovatieproject COCOON.

Couchbase, Inc. (2010). Couchbase. Retrieved from https://www.couchbase.com/

Couchbase, Inc. (2018). Couchbase and Apache CouchDB compared. Retrieved from https://www.couchbase.com/couchbase-vs-couchdb/

Dayley, B. (2014). *Node.js, MongoDB, and AngularJS web development*. Addison-Wesley Professional.

Dayne Hammes, H. M. & Mitchell, H. (2014). *Comparison of NoSQL and SQL Databases in the Cloud*. Southern Association for Information Systems.

Dean, J. & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*.

Free Software Foundation. (1985). Free Software Foundation. Retrieved from https://www.fsf.org/

Gartner, Inc. (2018a). 2018 Magic Quadrant for Operations Support Systems.

Gartner, Inc. (2018b). Magic Quadrant Methodology. Retrieved from https://www.gartner.com/technology/research/methodologies/research_mq.jsp

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

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

Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. (2013). Data Management in Cloud Environments: NoSQL and NewSQL Data Stores. *Journal of Cloud Computing: Advances, Systems and Applications*, *2*(1), 22. doi:10.1186/2192-113X-2-22

Härder, T. & Reuter, A. (1983). Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv. 15*(4), 287–317.

Hecht, R. & Jablonski, S. (2011). NoSQL Evaluation: A Use Case Oriented Survey. In *2011 International Conference on Cloud and Service Computing* (pp. 336–341).

Inc., M. (2009). MongoDB. Retrieved from https://www.mongodb.com/

Kaur, K. & Rani, R. (2013). Modeling and Querying Data in NoSQL Databases. In *2013 IEEE International Conference on Big Data* (pp. 1–7).

Kolomičenko, V., Svoboda, M., & Mlýnková, I. H. (2013). Experimental Comparison of Graph Databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services* (pp. 115–124). ACM.

Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *IEEE*.

Malda, R. & Bates, J. (1997). Slashdot. Retrieved from https://www.slashdot.org/

Maroo, T. (2013). *Handling with Dynamic, Large Data Sets - NoSQL a Buzzword or Savior?* JECRC Foundation.

Microsoft Corporation. (2002). LinkedIn. Retrieved from https://www.linkedin.com/

Miller, J. J. (2013). Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*.

Mohamed, M. A. & Ismail, M. O. (2014). Relational vs. NoSQL Databases: A Survey. *International Journal of Computer and Information Technology*, *3*(3).

Mohan, C. (2013). History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*.

MongoDB Inc. (2009a). MongoDB. Retrieved from https://www.mongodb.com/

MongoDB Inc. (2009b). MongoDB Sharding. Retrieved from https://docs.mongodb.com/manual/sharding/

MongoDB Inc. (2009c). Mongoid. Retrieved from https://docs.mongodb.com/mongoid/master/

Moniruzzaman, A. B. M. & Hossain, S. A. (2013). NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, *6*(4).

Nayak, A., Poriya, A., & Poojary, D. (2013a). Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, *5*(4).

Nayak, A., Poriya, A., & Poojary, D. (2013b, March). Type of NoSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems (IJAIS)*, *5*(4).

Neo Technology. (2007a). Neo4j. Retrieved from https://boltprotocol.org/

Neo Technology. (2007b). Neo4j. Retrieved from https://neo4j.com/

Open Webslides. (2017a, March). Open Webslides. Retrieved from http://openwebslides.github.io/

Open Webslides. (2017b). Open Webslides. Retrieved from http://openwebslides.github.io/

OrientDB Ltd. (2010). OrientDB. Retrieved from https://orientdb.com/

Paivio, A. (1969). Mental Imagery in Associative Learning and Memory. *Psychological Review*, *76*, 241–263.

Rohloff, K., Dean, M., Emmons, I., Ryder, D., & Sumner, J. (2007). An evaluation of triple-store technologies for large data stores. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*.

RubyGems. (2003). RubyGems. Retrieved from https://rubygems.org/

Sadalage, P. J. & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

Sanfilippo, S. (2009). Redis. Retrieved from https://redis.io/

Schmid, S., Galicz, E., & Reinhardt, W. (2015). Performance Investigation of Selected SQL and NoSQL Databases. In *AGILE 2015*.

Singh, G. (1996). Leader Election in the Presence of Link Failures. *IEEE Transactions on Parallel and Distributed Systems*, *7*(3), 231–236.

Solid IT. (2018). DB-Engine Ranking. Retrieved from https://db-engines.com/en/ranking

Technology, N. (2007). Neo4j. Retrieved from https://neo4j.com

The NoSQL Archive. (2018, April). The NoSQL Archive. Retrieved from http://nosql-database.org/

Underwood, B. (2010). Neo4j.rb. Retrieved from http://neo4jrb.io/

Upwork Global Inc. (2015). Upwork. Retrieved from https://www.upwork.com/

Weinberger, C. (2012). Is UNQL Dead? Retrieved from https://www.arangodb.com/2012/04/is_unql_dead/

Weinberger, C. (2016). Index Free Adjacency or Hybrid Indexes for Graph Databases. Retrieved from https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/

West, D. B. et al. (2001). *Introduction to Graph Theory*. Prentice hall Upper Saddle River.

Zhao, Y. (2015). *Event Based Transient Notification Architecture and NoSQL Solution for Astronomical Data Management* (Doctoral dissertation, Massey University).

Zhou, L., He, K., Sheng, X., & Wang, B. (2013). A Survey of Data Management System for Cloud Computing: Models and Searching Methods. *Research Journal of Applied Sciences, Engineering and Technology*, *6*(2), 244–248.