Linked data performance in different databases

Comparison between SQL and NoSQL databases

Prestanda med länkad data i olika databaser

Jämförelse mellan SQL och NoSQL databaser

ERICK CHAVEZ AND MANUEL MORAGA

Degree project, in
Computer Engineering,
First level, 15 hp
Supervisor at KTH: Reine Bergström
Examiner: Ibrahim Orhan
TRITA-STH 2014:67

KTH INSTITUTE OF TECHNOLOGY
School of Technology and Health
136 40 Handen, Sweden

Abstract

Meepo AB was investigating the possibility of developing a social rating and recommendation service. In a recommendation service, the user ratings are collected in a database, this data is then used in recommendation algorithms to create individual user recommendations.

The purpose of this study was to find out which demands are put on a DBMS, database management system, powering a recommendation service, what impact the NoSQL databases have on the performance of recommendation services compared to traditional relational databases, and which DBMS is most suited for storing the data needed to host a recommendation service.

Five distinct NoSQL and Relational DBMS were examined, from these three candidates were chosen for a closer comparison.

Following a study of recommendation algorithms and services, a test suite was created to compare DBMS performance in different areas using a data set of 100 million ratings.

The results show that MongoDB had the best performance in most use cases, while Neo4j and MySQL struggled with queries spanning the whole data set.

This paper however never compared performance for real production code. To get a better comparison, more research is needed. We recommend new performance tests for MongoDB and Neo4j using implementations of recommendation algorithms, a larger data set, and more powerful hardware.

Sammanfattning

Meepo AB undersökte möjligheten att utveckla en social betygs- och rekommendationstjänst. I en rekommendationstjänst samlas användarbetyg i en databas, för att sedan användas i en rekommendationsalgoritm för att skapa individuella rekommendationer till användarna.

Syftet med studien var att ta reda på vilka krav som ställs på ett DBMS, databassystem, som driver en rekommendationstjänst, vilken inverkan NoSQL-databaser har på prestandan för rekommendationstjänster jämfört med traditionella relationsdatabaser och vilket DBMS som är mest lämpat för användning i en rekommendation tjänst.

Fem olika NoSQL- och Relationsdatabaser undersöktes, från dessa valdes tre kandidater ut för en närmare jämförelse. Efter en studie i rekommendationsalgoritmer och rekommendationstjänster skapades en testsvit för att jämföra databasernas prestanda i olika områden. Till detta användes ett dataset med 100 miljoner betyg.

Resultaten visar att MongoDB hade bäst prestanda i flest användningsfall, medan Neo4j och MySQL hade problem med sökningar som sträcker sig över hela datasetet.

I denna uppsats jämförs dock inte prestandan med riktig produktionskod. För en bättre jämförelse behövs mer forskning. Vi rekommenderar nya prestandamätningar för MongoDB och Neo4j med implementationer av rekommendationsalgoritmer, ett större dataset och mer kraftfull hårdvara.

Table of Contents

1 Introduction

1.1 Problem description

Traditionally, most data has been stored in relational databases. Today, Relational Database Management Systems, or RDBMS, are the de facto standard of the industry. Most problems have been solved using a relational structure, without taking heed of what the data consists of and how it is used. Meepo AB wanted to develop a new rating and recommendation service, which would give users different recommendations depending on how they rated particular media. With the recent rise of NoSQL databases they wanted to know what impact a NoSQL solution would have on the performance of the database needed to drive a social recommendation service.

The database management system, or DBMS, is a crucial component of a recommendation service. The DBMS needs to scale with a growing user base, and allow changing the data model as the user base grows and the demands on the DBMS change.

This project investigated which database performs best in this particular use case, how NoSQL databases perform compared to traditional relational databases and which type of database is most suited for storing the data needed to host a recommendation service.

1.2 Goals

The goals consisted mainly of three parts, a case study, developing a testing skeleton and a comparative performance study.

1. A case study in which different characteristics of NoSQL and SQL databases are examined and compared. Also examine the use case of Meepo AB in more detail, together with the algorithms and queries needed for implementing a rating and recommendation service.
2. Developing a testing skeleton to use for the quantitative analysis. Use loose coupling between application layers to facilitate the replacement of the database layer implementation.
3. Quantitative analysis of the candidate DBMS using the testing skeleton, comparing the results to the characteristics from the case study.

1.3 Delimitations

This project was carried out for the purposes of Meepo AB. Thus, the particular use case of the company limited the scope of the study. All possible candidate databases were not examined due to time constraints, instead 3 main candidates were chosen for the comparison itself. There was no implementation of a rating and recommendation service available at the time, which meant that the queries being benchmarked are not real queries used in production. A real user recommendation service would run a recommendation algorithm. The actual queries needed to drive such an algorithm depend on the algorithm itself.

2 Theory

The first part of comparing different implementations of database management systems and their performance for use in recommendation services include a study of recommendation services and algorithms, followed by a study of NoSQL and SQL DBMS.

2.1 Recommendation data

Recommendation services are becoming increasingly popular, as they can lead to increased sales, by recommending new products to customers [1]. One of the first companies to implement a recommendation system is Amazon, which gives customers feedback on similar items in their store front. Streaming services such as Netflix also offer a recommendation service, to allow its customers to find new movies and television shows they want to watch, thereby increasing the value and usefulness of the service. The lessons learnt from the Netflix prize, a contest for bettering the collaborative filtering algorithm of Netflix [2], can be used to understand how such a service could be implemented.

2.2 Recommendation algorithms

Recommendation algorithms can be implemented in several different ways. Most algorithms have both benefits and drawbacks, particularly in how they cope with sparse data sets.

2.2.1 Item to item collaborative filtering

Item based collaborative filtering algorithms are based on calculating the similarity between two items. This similarity is calculated by comparing the items a user has a relationship to with other items in the data set. This similarity score is computed for each combination of item pair. When giving recommendations, this pair is looked up according to certain criteria e.g. recommending movies similar to one a user has just rated can be implemented by returning 5 movies with the highest similarity score. The similarity score itself can be calculated using different algorithms. Cosine-based similarity, Pearson correlation based in similarity and adjusted cosine similarity are some of the algorithms used for this calculation [3].

2.2.2 User to user collaborative filtering

User to user based collaborative filtering work in similar ways to the item based filters [3]. Here instead of calculating the similarity pairs of items, similarities are calculated for particular users. The similarity score is then used to find neighbors, which are users that have relationships to similar items, in this case, users that have rated similar movies. Users are then recommended movies that their neighbors like.

2.2.3 Singular value decomposition

Singular value decomposition or SVD is a form of matrix factorization as shown in equation (1), where M is a real or complex matrix of size m times n factorized into three matrices U, ∑ and V. The matrix ∑ is the one used in recommendation algorithms as it is a diagonal matrix of the size m times n containing the singular values of the matrix M. It can be used in recommendation algorithms by providing this decomposed matrix of user/movie pairs with an average recommendation value for each pair. This has been proven to give more accurate recommendations than collaborative filtering algorithms with dense data sets according to Sarwar et al [4].

$$UM = U\Sigma V^* \quad (1)$$

2.2.4 Rating & recommendation service

The Netflix prize [2] contest data consisted of the following: user, movie, date of grade, grade. The user and movie were integer ids, while the date of grade and grade were integer values.

A social rating and recommendation service for media will thus contain at the least the following entities:

- Users
- Ratings
- Media (Movies)

The users can rate different media, e.g. movies. Which are saved in a DBMS. Depending on the movies a user has rated and the rating score, the user should get recommendations for other similar movies. The recommendation themselves are powered through a recommendation engine, running a recommendation algorithm to determine which media to recommend, this algorithm and engine however are outside the scope of this study.

As the comparison has to be made without any algorithm, the focus is instead on the data itself, and the ability of different DBMS to query and traverse this data.

2.2.5 Connected data

The recommendation data is by design connected, and can be modelled as a graph as seen in figure 1.

Each user can rate many movies, and each movie can have many ratings. There is therefore a one to many relationship between movies and ratings, and between users and ratings.

To implement basic recommendations, the DBMS needs to connect this data by following the graph. Starting from a certain user, the users ratings are retrieved. Through the relationship between the ratings and the movies they are a rating of, other ratings of the same movie are retrieved. From these ratings, the users that have made the ratings are retrieved. Using this list of similar users, or neighbors, all the movies they have rated are retrieved. These can be aggregated, sorted and the most popular returned as recommendations.

Other queries found in most recommendation algorithms include calculations of a movies average rating, and other average scores taking into account appropriate constants.

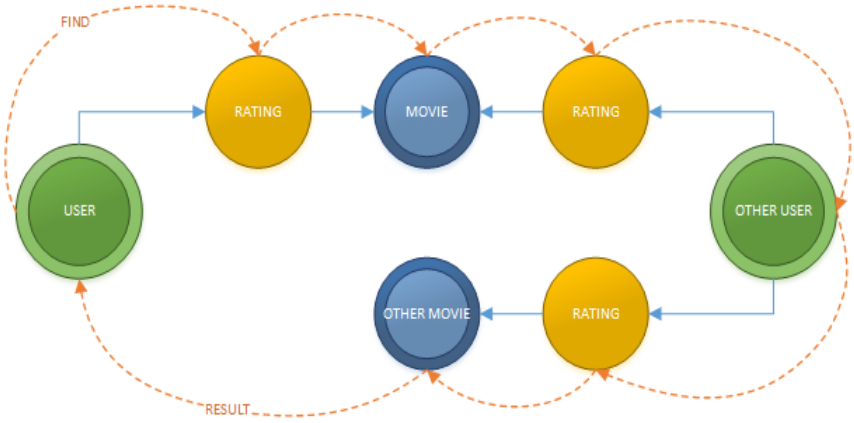

Figure 1: The image shows how the data is connected.

2.3 Database characteristics

Different databases have different capabilities, benefits and drawbacks. In choosing the correct database a specific set of characteristics thought necessary for implementing a rating and recommendation service were examined: how the database handles connected data, query speed with connected data, high availability, CRUD support, Index support, Query interface, schema and support.

2.3.1 Connected data

The data of a rating and recommendation service is by design connected. The different media is connected to ratings which are in turn connect to the user that created the rating. For a flexible model, it should be possible to add more connections between data: authors, actors, labels, friends and followers are just an example of other kinds of data that could be added in the future. Adding new kinds of data and new connections with it should be as trivial as possible.

2.3.2 Query speed

The speed of trivial queries is important for a web application. If the speed of trivial queries takes too long, this will have significant impact on latency of the client applications.

2.3.3 Query speed with connected data

Queries of connected data usually take longer than trivial queries. In traditional RDBMS they are made through joins, which can have a significant impact on performance. On other platforms, joins are not even possible, and the connectedness of data has to be handled explicitly in the application code.

2.3.4 CRUD support

Support for basic create, read, update and delete operations. Some database systems do not implement all CRUD operations, like Update, which means they have to be implemented in the application code.

2.3.5 Index support

Indexes can speed up queries by indexing entity fields that are often searched for. This often has a significant impact on performance.

2.3.6 Query interface

The query interface of the DBMS. As NoSQL DBMS do not use SQL for their queries, this means that developers need to learn a new query language to interact with the database. Some interfaces are more usable than others, and expose more commands and queries to the developer.

2.3.7 Schema

RDBMS traditionally use a schema with constraints that data has to conform to. Many NoSQL databases are instead schema-less and have no constraints on the data being entered into the database. This leaves the managing of constraints to the application which has both benefits and drawbacks.

2.3.8 Support

When developing a commercial application which will be used in production, support can be important when choosing a DBMS. Quality support, and the possibility of getting help when problems arise can even be a prerequisite for some companies.

2.4 NoSQL

Traditionally relational database systems have been used for persistence, and have become the de-facto industry standard. With the rise of web applications however, the volume of data, together with new requirements on availability and scalability was something traditional RDBMS could not cope with [5]. To meet this new demand, the NoSQL movement was born. NoSQL is a broad term, which encompasses several distinct kinds of database systems that have just one thing in common: they have left the relational model of tables and instead use different solutions for managing persistence in order to scale. NoSQL systems are designed to scale, and often do not adhere strictly to the ACID model of consistency to achieve this. The different kinds of NoSQL data models all have their strengths and weaknesses which have to be taken into account when choosing the appropriate tool for a certain task. Document Stores, Graph Databases and Wide Column stores were evaluated, and their suitability for modelling the data of a rating and recommendation service have been compared to the traditional Relational model.

2.4.1 Document stores

Document stores are a subset of NoSQL where data is saved as collections of documents instead of tables as in relational databases. A document database has no structured data, the data and all related data is grouped together and saved as a single collection with documents inside. This allows the document database to perform better when it comes to distributing information on several servers. Some document databases save data as JSON or as BSON.

JSON (JavaScript Object Notation) is an open standard for transmitting data objects as human readable text inspired by JavaScript objects. JSON objects contain a set of fields of name and value to represent data.

BSON (Binary JSON) is binary-encoded JSON, the difference is that BSON has more features for converting other languages to BSON and more type formats to use.

MongoDB

MongoDB is a c++ open-source project and one of the most popular document-oriented databases according to DB-Engines [6]. It is designed for use with distributed data and with large amounts of data, Big Data.

Documents are saved as BSON. The merging of JSON and binary-encoded format makes it more lightweight, flexible and makes it possible to match documents to queries.

The documents themselves consist of fields and values, separated by commas.

MongoDB features

Index

There are two index properties in MongoDB. Unique indexes, which create indexes only for the field if their values are not duplicated within the other values in the index list. Spare Indexes only index documents which contain the field that is being indexed, if the index field is empty then no index will be created for that document.

MongoDB documents by default always index the id field. Fields can also be indexed either as single fields, or as parts of a compound index. Compound indexes are made up of several fields which can only be queried together. Querying only one field of a compound index is not possible. There are also multikey indexes for arrays inside a document, geospatial indexes for 2 dimensional map coordinates and beta text indexes for searching strings. Hashed indexes are used in hashed shard keys for partitioning and distributing data on a shared cluster.

High Availability

MongoDB implements a replication process, where the data is duplicated in several database servers, data sets. The data sets have two types of priority, the primary data set that works with all write operations and the secondary data set. There are two secondary data sets that read changes from the primary data set, if the primary data set goes down then one of the secondary ones take the rank of primary, this operation makes the

data more available when errors occur and gives time for the broken primary data set to be fixed manually.

Sharding

The Sharding feature is crucial for how MongoDB manages large amounts of data, the incoming data is split by value onto other database servers and all this partition is made automatically by the auto sharding function. For example if the database server is dividing the data by alphabetical order and MongoDB is searching for some document with the N value, it only has to read from the database that contains the N-P values.

MapReduce

MapReduce is a process for aggregating the results from large amounts of data and was first introduced and explained by Google [7]. The mapReduce implementation tears down problems into smaller parts and aggregates the data. The first method, map() converts an amount of data with a key value to a key/value list for easy accessing in multiple clusters. The second function is reduce(), it takes the new key/value list, reduces it and puts the result in a collection. The MongoDB implementation of mapReduce has one more attached function, the finalize() function which makes it possible to make some final calculations on the result. The mapReduce function can return the result or make changes in the database.

Strengths

MongoDB is a great database for multiple applications, especially for object oriented applications. The greatest strength of MongoDB is the ability to handle large amounts of data. It was created for the new era of applications that require scalability, a flexible data model for agile development and to easily manage big data. The variety of indexes helps with optimizations of the aggregation speed for individual use cases. There a many commands that are similar to the SQL concepts and make it easy for new developers to adapt to MongoDB. MapReduce is not the only framework for doing advanced queries, there is also the Aggregation framework, a more easy way to do advanced queries. It is based on a pipes connection model, where the data can be managed in different steps.

Weaknesses

The weaknesses are side effects from the strengths of MongoDB.

The main weakness that is obvious from a relational database perspective is the collection restriction.

It is not supported to do joins so it is not possible to read from multiple collections at once.

It is very easy to save duplicated data in different collections if the data model design is implemented badly. This is a common problem in many databases but because MongoDB is more flexible with its data, anything can be put into documents. To prevent the input of malformed data, constraints and validation logic needs to be applied at the application level.

Because of the collection restriction and MongoDBs flexible data structure, it makes it more difficult to design a good data model for relational applications. Developers have created some model design patterns for relational use cases, which helps in some ways, but it is still hard to design data models for relational applications in MongoDB.

2.4.2 Graph databases

Graph databases are not new, but build upon graph theory used in mathematics. Many problems can be solved with graphs, particularly those that consist of networks, roads or other problems that can be modelled with a graph with many connections between its different nodes. Graph databases use this to build systems tailored to the graph model, and are in theory more suited for data containing many connections. Unlike relational database systems, graph database systems do not need any intermediary connections like intermediary many-to-many tables often used in RDBMS [8]. The relationships between nodes are instead stored directly as a physical property of the node itself. Graph database systems have gained an increased popularity with the rise of social media platforms, and are today used by market leading companies such as Facebook and Twitter [8]. There are currently two leading data models used in Graph databases, the property graph model and the resource description Framework. The property graph model is a shared model defined by the

Tinkerpop Blueprints framework [9]. It defines the graph as an object that contains vertices and edges. Both vertices and edges are elements which can have a set of properties stored as key/value-pairs. Vertices are objects that have incoming and outgoing edges, while Edges are objects with a tail and head vertex.

Neo4j

Neo4j is the most popular graph database today according to DB-Engines [10] and has been in use since 2007. It was chosen because of its relative maturity compared to other graph databases, while also being licensed under an open source license. It is written in Java and uses the property graph model. It stores its data in nodes connected by typed relationships. Values are known as properties and can be stored on both the nodes and the relationships themselves. Neo4j can be run as a server with a REST API or embedded into another application. It supports ACID transactions, indexes and distribution across multiple machines.

Features

Index

Neo4j supports indexes as traditional RDBMS do. Any property in the property graph can be indexed which leads to increased lookup performance. Indexes do however only speed-up the lookup of the initial startup nodes, they do not affect the speed at which the graph is traversed, as nodes are linked by relationships and not by IDs.

High Availability

High availability is supported under the Neo4j enterprise edition. It enables fault-tolerance through master-slave replication. It also provides horizontal scaling to make it possible for a system to handle more load than a single database instance would.

Optional schema

In Neo4j the schema is optional. This means that it can be used without any schema as other NoSQL database systems, or a schema can be implemented to gain the benefits of having one. This means that a service can be developed without any schema, which can then be added on later

in the development stage when the data model becomes more firm and different constraints can ease the use of the database.

Graph Algorithms

Through the graph algorithms component, Neo4j adds support for the following common graph algorithms: find shortest paths (using Dijkstra and A*), find simple paths, and find all simple paths. Dijkstra's algorithm is a common algorithm for finding single source shortest path trees [11]. The A* search algorithm is a heuristic extension to Dijkstra's algorithm [12]. Shortest path algorithms can be used in routing to find the shortest paths between network routers, locations on a map or friend connections in a social network.

Strengths

Neo4j is best suited for managing highly connected data [13]. Data that is highly connected, with different relationships between many different kinds of entities that can be modelled as nodes will be easy to traverse and query. Data that is self-contained without many relationships between entities is less suited for the Property Graph model.

2.4.3 Wide column stores

Wide Column store are NoSQL databases that mainly build on columns instead of rows as in relational databases. It is common to use wide column store databases as a Key/Value store. Among the wide column stores are Cassandra, HBase and Accumulo.

HBase

HBase is an open source column-oriented database written in Java, developed by Apache and based on BigTable, a high-performance database developed by Google and first described in a 2006 white paper [14]. The structure of HBase consists of tables, where each table contains column-families (groups of columns). The tables have a primary column with primary keys for selecting and gathering data. Queries are not supported as in other NoSQL databases. HBase runs on top of HDFS, the Hadoop Distributed System.

HBase and other parts of HBase like MapReduce, ZooKeeper and HDFS are developed by the Apache foundation using information gathered from the BigTable white papers published by Google.

HBase features

HBase is a high available database that can handle a large amount of data as it is implemented with Apaches ZooKeeper and runs on top of HDFS.

HDFS

HDFS is a distributed File System designed to handle large amounts of data, it uses File Blocks for storing data in multiple servers. A block can contain 64 MB or 128 MB.

HDFS has nodes (servers) inside a rack, many racks are grouped together and are called cluster. The data can be duplicated into blocks and distributed on several servers (nodes) inside a cluster.

Zookeeper

ZooKeeper is an open source server which helps coordinate distributed processes. Some common problems that zookeeper solves are race conditions, deadlocks, partial failures and coordination between many servers. This makes it possible for HBase to have numerous instances that are distributed on many servers.

Strengths

HBase is a powerful database when it comes to handling and retrieving huge amounts of data.

HBase has a flexible data structure, columns can be added whenever it is wished. Apache ZooKeeper and HDFS make HBase a very scalable and high-available database. ZooKeeper can manage the distribution on several clusters and HDFS is good at manage distributed data.

Weaknesses

The way data is retrieved is very limited. Only two commands are used to manage data: GET and PUT. GET is used to retrieve data and PUT is used to update or store data. The Data Model has to be designed before deploy-

ing an HBase database as it is not possible to change the structure of the columns after deployment.

2.5 Relational databases

Relational databases are currently the most popular and common DBMS. The main problem of this study was to investigate the performance of the new NoSQL database systems compared to these traditional Relational databases, making them an integral part of this investigation.

2.5.1 MySQL

MySQL is one of the most popular and widely used Relational DBMS on the market according to DB-Engines [8]. Because of it being released under an open source license it was chosen as the candidate for relational databases for this comparison. Because of its popularity, most developers today have used it at some point which makes finding people that know MySQL easy. MySQL was used as the standard against which the performance of the other database systems were compared.

2.5.2 NewSQL

NewSQL is a term given to Relational DBMS using the new technologies first introduced in the NoSQL-systems. Traditional Relational DBMS lacked support for usage in distributed systems which NoSQL solved by abandoning the relational model for simpler architectures. The term was first used after the release of Google's spanner whitepaper [15].

Some RDBMS do provide support for sharding data, ex MySQL, however there are no functions for easy handling of the different hosts in this distributed infrastructure.

NuoDB

NewSQL being a relatively new term, it does not have the same adoption as the somewhat more mature NoSQL databases. One of the most popular NewSQL database systems is NuoDB, which comes on the 47th place of the most popular relational systems on DB-engines [8]. It is designed from the beginning to offer a distributed database capable of cloud deployment while also exposing an SQL interface and the functionality expected from a traditional RDBMS, such as full ACID compatibility.

Three-tiered architecture

NuoDB is built on a three-tiered architecture: an administrative tier, a transactional tier (consisting of Transaction Engines) and a storage tier (consisting of Storage Managers). Traditionally, RDBMS have a tight coupling between the transaction and storage tier, bundling them together. According to the developers of NuoDB, decoupling these two tiers gives NuoDB increased scalability as the storage and transactional tiers can be scaled individually. The transactional and storage tiers communicate independently through peer-to-peer messaging.

Multi-version concurrency control

To handle consistency without blocking new reads through locks and deadlock detection, which can be detrimental to performance in a distributed system, NuoDB uses multi-version concurrency control. In this system all data is versioned, which means that the same data can be accessed independently on different hosts, and the version control system is used to resolve any conflicts that could emerge.

High availability

High availability is achieved by adding additional Transaction Engines and Storage Managers. NuoDB can scale-out to cover several separate data centers, making the same distributed database available across several geographic locations.

3 Design

Following the study of recommendation algorithms, recommendation services and DBMS, the different databases where evaluated. From these three candidates were chosen for the performance test. The data models for the three DBMS were designed, and a test application was developed in Java to test the performance of the three DBMS.

3.1 Evaluation

A comparison was made between the different DBMS according to their characteristics for connected data, query speed with connected data, high availability, CRUD support, Index support, Query interface, schema and support. These characteristics were summarized in table 1.

	MySQL	Neo4j	MongoDB	HBase	NuoDB
Database Type	Relational	Graph	Document	Wide Column Store	Relational
Connected data	Foreign key + joins	Graph model	Id:s (as foreign keys) + buckets. Joins at application level	Row keys as links in "edge" column family. Joins at application level	Foreign keys + join
Query speed (connected data)	Slower with increased data size (exponential decrease of performance)	Linear decrease in performance with increased data size	Fast in same collection. No queries between collections	Fast queries. No built-in queries for connected data	Slower with increased data size
High availability	No, only with MySQL cluster	In commercial version. Master-slave replication	Sharding + Master-slave replication	Zookeeper , Master-slave replication	In commercial version. Horizontal scaling
CRUD operations	Yes	Yes	Yes	get/put/delete	Yes
Index support	Primary and secondary index	Index on any attribute for labelled node	Index on any attribute. support for advanced indexes	Primary index	Primary and secondary index
Schema	yes	optional	no	no	yes
Query interface	SQL	Native Java, Gremlin, Cypher	JavaScript	JRuby	SQL
Support	Community support OR vendor support for commercial edition	Community support for community, personal, startups OR vendor support for enterprise edition	Community support OR vendor support for commercial edition	Community support OR Enterprise support through Hortonworks or Cloudera	Community support OR vendor support for proffessional edition

Table 1: Comparison between the databases in the study. The green field describes a desire property, yellow fields are cons but acceptable and red fields are cons.

3.1.1 Chosen candidates

MySQL

MySQL was chosen from the start as the DBMS to which the others would be compared. The relational model is a good fit for connected data, which it supports through using foreign keys. Queries for connected data can be made through join operations.

Neo4j

Neo4j has excellent support for highly connected data. The property graph model is a good fit for data which can be modelled as a graph. Data can be linked in many ways. Any node of any type can be linked to another using a new relationship. It is more flexible than other databases as new data models can be implemented alongside old ones, by drawing other kinds of relationships between nodes and assigning new labels. It has less support for high availability compared to the other DBMS, however this was not deemed as important as good support for connected data. Commercial support is available, and it provides indexes for speeding up lookups.

MongoDB

MongoDB supports highly connected data within a collection. It is not enough when data needs to be connected with or without a direct relation with each other. To achieve this in MongoDB, a good data model design and some logic in the application level must be implemented. But there are some downsides depending on the design pattern. The write performance can be high and the read performance can be really bad or vice versa. However the ability of MongoDB to handle large amounts of data, the flexibility of the data model and the fast access to data through different sorts of indexes are impressive features. The most important advantage of MongoDB is the flexibility to change the data model as desired for future development changes, if the important queries change then the data structure has to adapt to the queries and it is easily done by the shell that runs on JavaScript.

In conclusion MongoDB can be a great alternative to Neo4J and MySQL.

3.1.2 Not chosen

HBase

HBase is best suited for big data. Its scales well for very large amounts of data. This study is not aimed at measuring performance for data distributed between several servers, too large to fit on a single machine. This means that HBase loses its biggest strength. HBase itself does not provide means for querying connected data, all this logic has to be handled in the application layer, which was another reason for not including HBase in the performance tests.

However, there is also an option of using HBase as a persistence layer of an in-memory graph database like Titan. As Titan also uses Neo4j's property graph model, this can be a solution for data too big for Neo4j.

NuoDB

NuoDB has good support for highly connected data, being built on a relational model like MySQL. However the main difference between the two is NuoDB's abilities to scale. Running NuoDB on a single host would mitigate its strengths, making it less than ideal for this particular use case. It shares the ability of MySQL to query connected data through joins, and will probably behave in a similar manner to MySQL when it comes to query speed. It was therefore not chosen for this particular scenario.

3.2 Data model

3.2.1 MySQL

The data model of the MySQL implementation in practice followed the format of the Netflix dataset. Three tables were created: a movie table containing a "MOVIE_ID", "TITLE", and "YEAR". A "RATING" table containing a "RATING_ID", "SCORE", DATE, USER_ID and MOVIE_ID. The USER table contained only the USER_ID.

The user table and the rating table have a One-To-Many relationship, meaning that a user can have several ratings and for that reason a user id can be related to multiple rows in the rating table, the movie and the rating table also have a one-to-many relationship.

3.2.2 Neo4j

The data model of Neo4j followed the initial graph model described in chapter 2. The objects in this graph: users, movies and ratings became nodes in Neo4j. The relationship between these nodes were represented as rated and “has_rating” relationships respectively.

One gotcha of Neo4j is that the internal node IDs, which are used for looking up nodes are garbage collected. New nodes can therefore receive the IDs of old deleted nodes. This can be solved by using an indexed property set by the application as an ID instead, but means more metadata will be needed. Another solution is to use the internal node ID for lookups, and then use an application ID stored as a node property to make sure that the retrieved node is the same. This later method was used, to avoid the overhead of having duplicate indexed IDs.

3.2.3 MongoDB

MongoDBs data model has to be made depending on how the queries are constructed, which means that the information one is looking for has to be stored in a single collection. In this case all ratings are grouped by users and are saved in a “user_rating” collection, this sort of aggrupation is called ‘bucket’ and means that the relational data of an object is saved as an array in the main object/user so that relational queries can be made. This was necessary as it is not possible to make queries outside of a collection. Other objects are saved as common MongoDB documents containing only regular fields. All users are saved in a “user” collection and the movies in a “movie” collection. Compound indexes were created on movie titles and score together. Common indexes were created on rating score, movie title and reference ids. Implementation in Spring Data

For the performance tests, a service was written in Java to communicate with the databases. Spring Data was chosen as a tool, as it provided similar kinds of object relational mapping that exist for traditional RDBMS for both MongoDB and Neo4j. To help with making the coupling between the core business layer and the persistence layer a loose coupling, a hexagonal architecture was chosen instead of a traditional 3-layer architecture.

3.2.4 Architecture

The hexagonal architecture (or lifesaver architecture) consists of a core layer containing the business logic in the middle, with an integration layer containing implementations of external communications outside of this central layer. The core layer communicates with this outside layers through events. This solution is introduced to prevent the usage of business layer code in the outside layer dealing with particular implementations. In this particular use case, it meant that the persistence layer containing the different database implementations did not have any knowledge of the core business layer. At the same time, the core business layer did not have any knowledge of the implementation of the data persistence layer, which meant that the same core layer could be reused independently of which database was being tested. With this model, only the persistence layer implementation changed with the database.

The plain old java objects, POJOs for the user, rating and movie classes all had their id stored as strings. In this way, the database implementations could use any object best suited to represent the database id, while the resulting object passed to the core layer would always contain a string id, independently of the database being used.

Spring Data JPA

To connect the benchmarking application to MySQL, Spring Data JPA was used. Repositories were created for the simple CRUD operations with the user, rating and movie tables. In Spring Data, this is easily handled through creating a Repository Interface, containing the different methods that will be used to access the repository. The implementation is then produced by Spring Data.

For the more advanced queries of connected data, custom implementations of these repositories were written. In Spring Data, this is done by creating a new Interface describing the new methods. These methods are then implemented in a repository implementation.

The objects were mapped with JPA annotations to handle the mapping of Java objects to database tables.

Queries

Two queries were written to test the performance of connected data. The first one queries the database for similar users, in this case it means users that have rated the same movies as the user id that is supplied to the query.

The second query is an extended version of the first one. Here, the answer from the first query is joined with the rating table again, to find which movies that these similar users have rated and the user has not. The movies are grouped and sorted by count.

```
SELECT
    S.USER_ID
FROM
    RATING F
        INNER JOIN
    RATING S ON F.MOVIE_ID = S.MOVIE_ID
WHERE
    F.USER_ID = $1 AND S.USER_ID <> $1;
```

Figure 2: Find similar users

```
SELECT
    COUNT(T.MOVIE_ID), T.MOVIE_ID
FROM
    RATING F
        INNER JOIN
    RATING S ON F.MOVIE_ID = S.MOVIE_ID
        INNER JOIN
    RATING T ON S.USER_ID = T.USER_ID
WHERE
    F.USER_ID = $1 AND S.USER_ID <> $1
        AND T.MOVIE_ID <> F.MOVIE_ID
GROUP BY T.MOVIE_ID
ORDER BY COUNT(T.MOVIE_ID) DESC;
```

Figure 3: Shows movies rated by similar users grouped by movie count.

Spring data neo4j

As with the JPA implementation, the Neo4j implementation in Spring Data Neo4j contained Spring Data repositories for handling simple CRUD operations. Again, for the advanced queries, these Repositories were extended with Implementations of new services for more advanced queries.

The domain objects mapping the Neo4j nodes to Java objects were annotated using Neo4j annotations. These were quite similar to the Neo4j domain objects, the main difference was that the id field in the Neo4j implementation represents the internal neo4j node id. As previously noted when discussing the data model, an appId was needed to guarantee that retrieved nodes were the same ones that had been previously persisted. The two id and appId fields were then used together to identify a particular node. When communicating with the core layer, these two fields were passed in as a single string id.

Neo4j has several alternatives for querying the database. The queries were written in Cypher as it assembles SQL the most. The database itself was run as an embedded database, as this should give the best performance [13]

Queries

In Cypher, queries are made by specifying a starting node. This node can then be matched to other nodes by describing the relationships between them, and the nodes they connect to.

The queries that had been written in SQL were translated into Cypher, to get a comparison that was as equal as possible. Writing these queries was simpler than SQL, and the resulting Cypher queries were shorter.

Similar users:

$$start\ u = node(\{id\})match(u) - [:RATED] ->\ (r:Rating) <- [:HAS\_RATING] - (m:Movie) - [:HAS\_RATING] -> (r2:Rating) <- [RATED] - (u2:User)\ return\ u2;$$

Similar movies:

$$start\ u = node(\{id\})match(u:) - [:RATED] \rightarrow (r:Rating) <- [:HAS_{RATING}] - (m:Movie) - [:HAS_{RATING}] \rightarrow (r2:Rating) <- [RATED] - (u2:User) - [:RATED] -> (r3:Rating) <- [:HAS\_RATING] - (m2:Movie)\ return\ m2;$$

Spring Data MongoDB

MongoDB has an aggregation framework designed for advanced queries but then again it has its own limitations. The aggregate query have a limit of 100MB that can be passed through its pipes within the query and the results can't be bigger than 16MB. However in the newer version releases, from 2.6 and newer, the result/data can be held on disk which removes these limitations.
Spring Data helps a lot with advanced queries and object mapping, how-

ever it does not support everything in the MongoDB aggregation framework. The option to aggregate a query using disk storage is still not supported. Therefore Spring Data was only used for implementing CRUD repositories, as for the advanced queries the native Java Driver from MongoDB was the only option.

Queries

As in Neo4j, similar queries has been created in MongoDB to achieve the same result. In Java the MongoDB aggregation queries are noticeably longer than similar queries in Neo4j.

```
db.user_rating.
    aggregate([
        {$match:{
            "USER_ID":{"$nin":[6]},
            "ratings":{$elemMatch:{
                "MOVIE_ID":{$in:movieRefArray},
                "SCORE":{"$gte":5}}}}},
        {$unwind: "$ratings"},
        {$match:{
            "ratings.MOVIE_ID":{
                "$nin":movieRefArray}}},
        {$group:{
            _id:{MOVIE:"$ratings.MOVIE_ID"},
            MOVIE_COUNT:{"$sum":1},
            TOTAL_SCORE:{"$sum":"$ratings.SCORE"}}},
        {$sort:{"MOVIE_COUNT":-1,"TOTAL_SCORE":-1}},
        {"$limit":1000}],
        {allowDiskUse: true}).pretty()
```

Figure 4: Native query for finding all movies by id from a user

```
//new empty array
var movieRefArray = []
//push ids from the users movies to an array
db.user_rating.
    aggregate([
        {$match:{"USER_ID":6}},
        {$unwind:"$ratings"},
        {$match:{"ratings.SCORE":
            {"$gte":3}}},
        {$sort:{"ratings.SCORE":-1}},
        {$project:{MOVIE_ID:"$ratings.MOVIE_ID"}}
        ],{allowDiskUse: true}).forEach(
            function(x){
                movieRefArray.push(x.MOVIE_ID)
                    }
            )

```

Figure 5: Native query for finding all the similar movies by id from a user

3.2.5 Data migration

Netflix Data

The Netflix dataset contains the database data as comma separated values, CSV text files. The text is formatted as tables with rows of values where every value is separated with a comma (,):

Ratings: There are 177771 ratings files were each file contains all ratings for a single movie. The first line of a rating file contains the id of the movie followed by a colon (:). Each rating line contains the user id, the score with a scale from 1 to 5 and the date in the format YYYY:MM:DD.
Movies: There is a "movie_titles" file that contains every movie with its title, year of release and movie id.
Other files: There are more text files explaining how the data looks like and the quantity of users, movies and ratings, the ratings range and other instructions.
The Netflix dataset was extracted from a relational database therefore it was easiest to migrate the dataset to a MySQL database and then migrate the data from MySQL to the other the databases in the appropriate format.

MySQL

The Netflix dataset was imported to MySQL with a script written in the Ruby programming language.

A user text file from the Netflix dataset did not exist yet the user table was created with only a user id column. The user id was extracted from the rating table with a query only asking for all distinct user ids from the rating table.

Neo4j

The test data was imported into Neo4j using the Neo4j Batch Importer. As it accepts text files with tab separated values, these were exported from the MySQL tables.

Relationships and nodes are imported in separate files, the nodes themselves can be linked either by their node id or indexed value. Linking

nodes using indexed values led to memory errors when linking the RATED relationship between Users and Ratings. The internal node IDs were used instead, as these are assigned in order of insertion, the MOVIE_ID and RATING_ID from the MySQL database were used with an offset to connect the nodes with the HAS_RATING relationship. The users however, do not contain sorted IDs as the USER_ID are just values in the RATING table. To import these, a new USER table was created in the MySQL database containing all distinct USER_ID values. To this table a separate column with auto incremented IDs was added. Next, an update query added a new column to the RATING table, referencing the new ordered ID of the USER_ID table. The RATED relationship could then be exported from the RATING table, and after adding an offset to match the node IDs of the User node, they were imported into the Neo4j.

MongoDB

MySQL can export data as CSV files from its queries. This was helpful for creating CSV files with the desired data format which could later then be migrated to both MongoDB and Neo4j.

MongoDB has a simple import command function for migrating text files, sadly it does not support importing arrays or other advanced objects into MongoDB. JavaScript is used in the MongoDB command shell, as it was the easiest way to manipulate the data, all data was imported to MongoDB from MySQL through CSV files as simple documents without arrays. Then the data was reorganized using the command shell. Buckets were created for each user containing all ratings made by that user.

4 Results

Using the test application developed for the DBMS

4.1 Benchmark framework

The benchmarks were run using caliper, a tool originally developed by Google for microbenchmarks. The benefits of using caliper instead of simply measuring start and end times using System.nanoTime() is that it helps with several things that are important in benchmarking Java Code. Benchmarks can be performed in a more controlled, standardized way. By tracking all options sent to the JVM, differences between test setups can be identified to avoid benchmarking with different options. It also simplifies the warm up needed to make sure that the JIT compiler has already performed most optimizations, to avoid the overhead of JIT compilation during benchmarks. Each test is run 9 times, following the warm-up phase. In the warm-up phase, the test runs for 5 minutes and all results measured during this phase are discarded. The mean value of the test results is then returned.

4.2 Database server

The benchmarks were run on VMware server virtual machine instances. Each virtual machine test server had the following specifications.

Server properties	
Total Physical Memory	8178592 KB
CPU	Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz x 2
Java vm	Oracle Corporation
Java runtime version	1.8.0-b132
OS.	Ubuntu 12.04 LTS amd64
Linux Kernel	3.2.0-61-generic

Table 2: properties of the benchmark server

4.2.1 Queries

Seven different queries were benchmarked, to test the performance of the DBMS. Some queries are more simple retrieval queries, while other queries are more complicated queries, needing more computing power and longer execution times. Table 3 shows seven queries with difficulty and criteria. The difficulty color indicates how difficult and challenging every query is based on the criteria of the result. Green color are supposed to give a faster result than the other difficulty colors, the orange color are queries with demands on the result and the red are queries that are supposed to take a high amount of computer resources .

Difficulty	Queries	Criteria
QUERY 1	Find ratings by user id	Sorted by date
QUERY 2	Find movies rated by user	Sorted by score
QUERY 3	Count movie ratings	none
QUERY 4	Find movies that have most reviews with the highest score	Sorted by total ratings (limit 100)
QUERY 5	Find the most popular movies from all the other users who also have rated the same movie as the main user	Collect only rated movies equal and greater than score 4, then sorted by quantity (Limit 100)
QUERY 6	Find Movie by id	None
QUERY 7	Calculate the average of a movie score	None

Table 3: The seven benchmarked queries

Query 1 (Find ratings from a user)

The purpose of query 1 is to test the query speed when retrieving all the ratings done by a user. The result is only the list of ratings ordered by ascending date.

MySQL behavior

$$SELECT * FROM\ rating\ WHERE\ USER\_ID = (id)\ ORDER\ BY\ DATE;$$

To achieve the result MySQL only reads from an indexed field USER_ID on the table rating. The result only extracts the ids of the user and movie.

Neo4j behavior

$$START\ n = node(\{nodeId\}) WHERE\ n.appId = \{appId\} MATCH\ (n) - [:RATED] \rightarrow (r:Rating)\ RETURN\ r\ order\ by\ r.date\ DESC$$

MongoDB behavior

$$db.user\_rating.aggregate([\ \{\$match: \{"USER\_ID": (id)\}\},\ \{\$unwind:\$ratings\},\{\$sort: \{ratings.DATE: -1\}\},\ \{\$group: \{"\_id": "\$USER\_ID", "ratings": \{\$push: "\$ratings"\ \}\}\}],\ \{allowDiskUse: true\})$$

MongoDB Aggregate query searches the user by id within the user_rating collection, it splits the ratings to individual ratings, sorts them by date and then it recreates the array, the returned new array is sorted by date.

This time the result did not return the rating list with its movie a user as Neo4j did, only ids were retrieved for user and movie. The query time would be longer if every movie and user were extracted from the database and returned with each rating.

Result: MongoDB got the best result but the result only returned ids, date and score, while Neo4j also retrieved the movies title and the year.

Query 1	MySQL	MongoDB	Neo4j
Result	5.61 ms	4.00 ms	6.59 ms

Table 4: Result of query 1.

Query 2 (Find rated movies from a user)

The result contains all the rated movies from a user, sorted by descending score. This query tests the performance of connecting a user to its rated movies.

MySQL behavior

$$SELECT\ m.MOVIE_{ID}, m.TITLE, m.YEAR$$
$$FROM\ rating\ r\ INNER\ JOIN\ movie\ as\ m$$
$$ON\ m.MOVIE_{ID} = r.MOVIE_{ID}$$
$$WHERE\ r.USER_{ID} = (id)$$
$$ORDER\ BY\ SCORE\ DESC;$$

To retrieve the result in MySQL, two tables need to be joined, the rating and movie table. From this the movie information is extracted in descending order.

Neo4j behavior

$$start\ u = node(\{nodeId\})$$
$$where\ u.appId = \{appId\} match\ (u)$$
$$-[:RATED] \rightarrow (r)$$
$$< -[:HAS_{RATING}] - (m)$$
$$return\ m\ ORDER\ BY\ r.rating\ desc$$

MongoDB behavior
Query part 1:

```
db.user_rating.aggregate([
{$match:{"USER_ID":6}},
{$unwind:$ratings},
{$sort:{ratings.SCORE:-1}},
{$group:{_id:$USER_ID,ratings:{$push:$ratings}}},
{$project:{"_id":0,"MOVIE_ID":
"$ratings.MOVIE_ID"}} ],
{allowDiskUse:true})
```

Query part 2:

```
db.movie.aggregate( [
{$match:{"_id": {$in:(listOfmovieIds)}} } ],
 {allowDiskUse:true})
```

The query part 1 and the query 2 are similar to each other, the difference is that the query 2 only returns the movies ids from the database.

Query part 2 takes an array of ids and returns all the movies with their titles, Year and id.

Result: This time every database returned the same amount of data and the database with best result is Mysql.

QUERY 2	MySQL	MongoDB	Neo4j
Result	4.71 ms	7.20 ms	6.04 ms

Table 5: Result of query 2

Query 3 (Find all users that rated a movie id)

The result is the total of users that rated a movie and the purpose of this query is to test the ability to count fields.

MySQL behavior

```
SELECT COUNT(USER_ID)
FROM rating
WHERE MOVIE_ID = (id);
```

MySQL has a COUNT function and it is simple to use.

Neo4j behavior

```
START n = node({nodeId})
WHERE n.appId = {appId}
MATCH (n) − [r: HAS_RATING] − () RETURN count(r);
```

MongoDB behavior

```
Query: db.user_ratings.
find({"ratings.MOVIE_ID" : (id)}).
count()
```

Here the Aggregate framework wasn't needed, it was only necessary to make a common find query and call the count function.

Result: MongoDB got the best time and returned the same result as the other databases.

QUERY 3	MySQL	MongoDB	Neo4j
Result	3.07 ms	0.59 ms	0.99 ms

Table 6: Result of query 3

Query 4 (Find movies that have most reviews with the highest score)

This query was chosen due to its difficulty and the big amounts of data handled when calculating the most popular movies.

MySQL behavior

```
SELECT m.MOVIE_ID, m.TITLE, m.YEAR,
( SELECT COUNT(*) FROM rating r2
WHERE r2.MOVIE_ID = r.MOVIE_ID ) AS count
FROM rating AS r INNER
JOIN movie as m ON r.MOVIE_ID = m.MOVIE_ID
WHERE r.SCORE = 5 GROUP BY r.MOVIE_ID
ORDER BY count DESC;
```

Each movie is counted in order of appearance, and joined with the movie table to get the movie properties. The result is sorted by count.

Neo4j behavior

```
MATCH (m:Movie) - [:HAS_RATING]
→ (r:Rating) where r.rating = 5 with m, count(r) AS c
RETURN m, c
ORDER BY c DESC LIMIT 1000;
```

MongoDB behavior

Query part 1:

```
db.user_rating.aggregate([
{$match: {USER_ID: 6}},
 {$unwind:$ratings},
{$match: {ratings.SCORE: {$gte: (4)}}},
{$project: {MOVIE_ID: "$ratings.MOVIE_ID"}}],
{allowDiskUse: true})
```

Query part 2:

```
db.user_rating.aggregate([
{$match: {"USER_ID": {$nin: [(6)]},"ratings":
{$elemMatch: {"MOVIE_ID":
{$in: (movieRefArray)}, "SCORE": {$gte: 4}}}}}, {$unwind: "$ratings"},
 {$match: {ratings.MOVIE_ID: {$nin :
movieRefArray}}},
{$group: {_id: {MOVIE: "$ratings.MOVIE_ID"},
 MOVIE_COUNT: {$sum: 1}, TOTAL_SCORE:
{$sum: "$ratings.SCORE"}}},
{$sort: {MOVIE_COUNT: −1, TOTAL_SCORE: −1}},
{$limit: 1000}], {allowDiskUse: true})
```

Query part 3:

```
db.movie.aggregate( [
{$match: {"_id":  {$in: (listOfmovieIds)} } } ],
{allowDiskUse: true})
```


It requires 3 operations to gain the same result of objects in MongoDB, the first part of the query is to return all the main users movies ids. The second Query part 2 retrieved a list of top hundred movies ids within all the other users that have rated high on the main users movies. The last operation extracted the movie information from the list gained from query part 2.

Result: MySQL and Neo4j did not complete the task. The MySQL query process was shutdown after 10 hours, while Neo4j crashed as it ran out of memory. MongoDB finished the query in 4 minutes and it made it thanks to the ability to write the data to disk while using the aggregation query.

Query 4	MySQL	MongoDB	Neo4j
Result	Does not complete	241255 ms (4.0 min.)	Does not complete

Table 7: Result of query 4.

Query 5 (Find the most popular movies from all the other users who also have rated the same movies as the main user)

The result is a list of movies that were collected from all the other users that have also rated the same movies as the main user and it is limited by 100 movies. This query test the ability to connect a large amount of entities and aggregating them.

MySQL behavior

```
SELECT COUNT(T.MOVIE_ID),T.MOVIE_ID
FROM rating F INNER JOIN rating
ON F.MOVIE_ID = S.MOVIE_IDINNER
JOIN rating T ON S.USER_ID = T.USER_ID
WHERE F.USER_ID = (id)AND S.USER_ID <> (id)
AND T.MOVIE_ID <> F.MOVIE_ID
GROUP BY T.MOVIE_ID
ORDER BY COUNT(T.MOVIE_ID) DESC;
```

Neo4j behavior

```
START u = node({userId})WHERE u.appId =
{appId}MATCH (u) − [:RATED] → (r) <
−[:HAS_RATING] − (m:Movie) − [:HAS_RATING] →
(r2:Rating) < −[RATED] − (u2:User) − [:RATED] →
(r3) < −[:HAS_RATING] −
(m2:Movie)return count(m2),m2
ORDER BY count(m2) DESC LIMIT 100;
```

MongoDB behavior

Query part 1:

```
db.user_rating.aggregate([
{$match: {"USER_ID": (id)}},
{$unwind:$ratings},{$match: {ratings.SCORE: {$gte: 4}}},
{$project: {MOVIE_ID: "$ratings.MOVIE_ID"}}],
{allowDiskUse: true})
```

Query part 2:

```
db.user_rating.aggregate([
{$match: {"USER_ID": {$nin: [(id)]},
"ratings": {$elemMatch: {"MOVIE_ID":
{$in: movieRefArray}, "SCORE": {$gte: (4)}}}}}, {$unwind: "$ratings"}, {$ma

{$nin: movieRefArray}}}, {$group: {_id: {MOVIE: "$ratings.MOVIE_ID"},
MOVIe_sum: {$sum: 1}, TOTAL_SCORE:
{$sum: "$ratings.SCORE"}}},
{$sort: {MOVIE_COUNT: −1, TOTAL_SCORE: −1}},
{$limit: 1000}], {allowDiskUse: true})
```

Query part 3:

```
db.movie.aggregate( [
{$match: {"_id": {$in: (listOfmovieIds)} } } ],
{allowDiskUse: true})
```

In MongoDB a three query operation is needed to achieve the result, the first operation, returns a list of movies ids from the users rating-bucket. In the second query, the top hundred movies are calculated and returned a list of movies ids. The third part collects all the Movie information from the list of movies ids.

Result: With Query 4 and 5 the only DBMS that returns a result is MongoDB with an aggregate query. MongoDB returns the result after 4 minutes. As Neo4j can't load the data into RAM, it crashes. MySQL did complete the query after running for 10 hours.

Query 5	MySQL	MongoDB	Neo4j
Result	Does not complete	241140 ms (4.01 min.)	Does not complete

Table 8: result of query 5

Query 6 (Find Movie by id)

MySQL behavior

```
SELECT * FROM movie WHERE MOVIE_ID = (id);
```

Neo4j behavior

```
START n = node({nodeId})
WHERE n.appId = {appId} RETURN n;
```

MongoDB behavior

```
db.movie.find({"_id": (id)})
```

Result: Neo4j was the fastest. Neo4j time twice as fast as MongoDB, while MySQL was the slowest.

Query 6	MySQL	MongoDB	Neo4j
Result	1.98 ms	0.24 ms	0.12 ms

Table 9: Result time of query 6

Query 7 (Calculate the average movie score)

Query 7 is a query sometimes used in recommendation algorithms, this query tests the ability to calculate the average result of entity properties.

MySQL behavior

```
SELECT avg (MOVIE_ID)
FROM rating
WHERE MOVIE_ID = (id);
```

MySQL has a simple average function, avg().

Neo4j behavior

```
START n = node({nodeId})WHERE n.appId
= {appId}MATCH (n) − [:HAS_RATING]
−> (r:Rating) RETURN avg(r.rating)
```

MongoDB behavior

```
Query: db.user_rating.aggregate([
{$match:{ratings.MOVIE_ID:1}},
{$unwind:$ratings},
{$match:{ratings.MOVIE_ID:1}},
{$group:{"_id":"$ratings.MOVIE_ID","ratings":
 {$avg:"$ratings.SCORE"}}}],
{allowDiskUse:true})
```

MongoDB have an average method, however as the ratings are implemented as user buckets, the aggregate query needs to collect all user buckets that include the movie id. It then splits them into separate ratings and removes the unwanted ratings of other movies. The average score of the movie is then calculated.

Results: Neo4j had the best result. MongoDB result was slow due to the unwound array it had to do when it only wanted to extract all the ratings of a movie. MySQL did only a simple average query but got sixteen times slower time than Neo4j.

Query 7	MySQL	MongoDB	Neo4j
Result	31.61 ms	562.12 ms	1.82 ms

Table 10: Result time of query 7

Result summary

Table 11 shows the compiled results from all queries.

	MySQL	MongoDB	Neo4j
Result(Query 1)	5.61 ms	4.00 ms	6.59 ms
Result(Query 2)	4.71 ms	7.20 ms	6.04 ms
Result(Query 3)	3.07 ms	0.59 ms	0.99 ms
Result(Query 4)	Does not complete	241255 ms (4.0 min.)	Does not complete
Result(Query 5)	Does not complete	241140 ms (4.01 min.)	Does not complete
Result(Query 6)	1.98 ms	0.24 ms	0.12 ms
Result(Query 7)	31.61 ms	562.12 ms	1.82 ms

Table 11: The result of all the queries.

5 Discussion

The performance results show that MongoDB had the best performance in most categories compared to MySQL and Neo4j. However it is important to remember that the queries being tested are not real recommendation algorithms used in production. The queries are mostly small queries that are sometimes used in recommendation algorithms. To get a more accurate comparison of the DBMS the databases would have to be compared using one or more implementations of a real recommendation algorithm. This was however outside the scope of this study.

MongoDB struggles when the data being queried lies in different collections. However, a well implemented data model using buckets for connected data that needs to be retrieved often resulted in better performance than both Neo4j and MySQL. Through the usage of functions like the aggregate framework or MapReduce, together with the aggregation of connected data into buckets, Mongo can handle large amounts of data well. Both Query 4 and 5, which Neo4j and MySQL didn't complete, were completed by Neo4j in 4 minutes. The retrieval of individual nodes is very fast and Mongo therefore performed best overall compared to the other DBMS. The absence of schema means that the data model could be changed easily. This was shown when importing data from the MySQL tables. The data was imported as is, and later refactored into the desired data model. The query interface is more advanced than the other DBMS as it implements functions like MapReduce. MapReduce was not used for any queries in the performance tests, but could prove to be an efficient way of implementing a recommendation algorithm that handles big data sets.

Neo4j is shown to have really bad performance with queries that handle large amounts of data. As a queries must fit into one transaction, those that are too large will result in out of memory errors. If a query has to traverse the whole graph, or calculate values that depend on traversing the whole graph (e.g. traversing all ratings, or all users) too many nodes will be loaded into the DBMS and an out of memory exception will occur. Unlike RDBMS it is not possible to limit the search by table row number,

which means that performing a big query in small iterations is much harder. This could be achieved through the usage of additional labels or indexed ids set by the application. According to the neo technology hardware sizing calculator, the recommended hardware for running neo4j with the Netflix dataset are at least 4 CPU cores and around 23G of RAM. However this was not available at the time. The performance of Neo4j could change dramatically if the RAM recommendations were followed, as large queries containing big parts of the graph would then fit into memory. The performance when accessing limited parts of the graph however was good. In Query 7: Calculating the average rating for a movie, Neo4j outperformed both MongoDB and MySQL. Still, this shows that Neo4j is not suited for queries spanning the whole data set. When all nodes have to be visited, a document structure is more efficient than a graph.

The flexible schema of Neo4j proved valuable when developing the testing skeleton. Labels and new relationships can be used to implement new data models or speed up queries that are performed often. The Cypher query language turned out to be simpler to use for more complex queries than SQL.

MySQL had decent performance for most queries being benchmarked. However it struggles with big joins. Neither Query 4 nor 5 did complete, even after several hours, while the same queries in MongoDB took 4 minutes. MySQL did complete the same queries with smaller data sets, but struggled with the testing data set of 100 million ratings. This shows that joining big tables is not effective. MySQL is not flexible when it comes to the data model. Running Alter Table statements to change the data model is much more time intensive than data model changes are in Neo4j and MongoDB. Advanced queries consisting of several joined tables are complicated to write, and prone to errors. In this area, MySQL was deemed much harder to use than the other DBMS. However, given the popularity of SQL databases, finding developers that know SQL is much easier.

5.1 Impacts on economic, social and environmentally sustainable progress

If sustainable economic progress is defined as increased profitability, then recommendation services do have a positive impact on the number of sales and therefore profits. This work has therefore contributed to more efficient use of resources. It is however not possible to draw any conclusions on the environmental impact of this work.

With access to personal user data, services have a great responsibility in maintaining this data private. Any data collected should be done so at the discretion of the user, and used only according to the terms accepted by the user. This can however sometimes conflict with the interests of the research community.

During the Netflix Prize contest, Netflix made parts of their user data available to the contestants. This data had been scrambled, to avoid the identification of individual users and their private recommendation data. However some researchers showed that it was possible to retrieve the original user data. Following this, Netflix was sued by 4 users. A follow up contest was cancelled because Netflix could not guarantee that this exposal of user data would not happen again. If data from a recommendation service will be used for research purposes, the user has to be informed of this from the start. Great care has to be taken into obfuscating the data to make it less probable that a user can be connected to his or hers personal data.

6 Conclusions

In this work a case study was performed to examine the different characteristics of NoSQL and SQL databases, together with the use case of Meepo AB and algorithms and queries needed to implement a recommendation service. A testing skeleton was developed for a quantitative analysis between the DBMS. It used the hexagonal architecture to make the coupling between application layers loose. A quantitative analysis was performed, where performance tests were run to compare the DBMS. The results were then discussed and compared to the different criteria set during the case study.

The performance tests show that MongoDB seems like the best choice for this particular use case, using this particular data set and hardware configuration.

Neo4j suffers from not being able to load the whole graph into RAM, while MySQL performs average but seems to have problems with larger data sets. However, this shows that graph databases such as Neo4j are not the optimal choice for queries that need to access the whole database. It also showed that NoSQL databases have several advantages compared to traditional SQL databases. This work has shown that compared to both MySQL and Neo4j, MongoDB seems like a better choice for storing data needed to host a recommendation service.

However it is difficult to say what impact this would have on a real production environment. For this, more research is needed. It would be desirable to perform new performance tests using real recommendation algorithms, more RAM to allow Neo4j to load the whole graph into memory and bigger data sets to see how the DBMS would handle scaling to several servers.

References

[1] P.-Y. a. W. S.-y. Chen, "Does Collaborative Filtering Technology Impact Sales? Empirical Evidence from Amazon.Com" *Social Science Research Network,* 2007.

[2] J. B. a. S. Lanning, "The Netflix Prize," *KDD Cup and Workshop,* 2007.

[3] G. K. J. K. a. J. R. Badrul Sarwar, " Item-based collaborative filtering recommendation algorithms.," *WWW10,* 2001, Hong Kong.

[4] B. M. S. a. G. K. a. J. A. K. a. J. T. Riedl, "Application of Dimensionality Reduction in Recommender System -- A Case Study," *WebKDD-2000 Workshop,* 2000, Minneapolis.

[5] G. Burd, "SYSADMIN NoSQL," 2011.

[6] DB-Engines, "DB-Engines Ranking of Document Storage," May 2014. [Online]. Available: http://db-engines.com/en/ranking/document+store. [Accessed 13 06 2014].

[7] S. G. Jeffrey Dean, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation,* 2004.

[8] DB-Engies, "DB-Engines Ranking of Relational DBMS," DB-Engies, 05 2014. [Online]. Available: http://db-engines.com/en/ranking/relational+dbms. [Accessed 23 05 2014].

[9] M. A. Rodriguez, "Property Graph Model," 13 Jun 2012. [Online]. Available: https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model. [Accessed 27 05 2014].

[10] DB-Engines, "DB-Engines Ranking of Graph DBMS," 05 2014. [Online]. Available: http://db-engines.com/en/ranking/graph+dbms. [Accessed 23 05 2014].

[11] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* vol. I, no. 1, pp. 269-271, 1959.

[12] P. N. N. R. B. Hart, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics,* vol. IV, no. 2, 1968.

[13] F. H. a. R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j" *In Proceedings of the Joint EDBT/ICDT 2013 Workshops,* no. EDBT 13, 2013, Genoa,.

[14] J. D. S. G. W. C. H. D. A. W. M. B. T. C. A. F. a. R. E. G. Fay Chang, "Bigtable: A Distributed Storage System for Structured Data," *OSDI'06: Seventh Symposium on Operating System Design and Implementation,,* 2006.

[15] J. D. M. E. A. F. C. F. J. F. S. G. A. G. C. H. P. H. W. H. S. K. E. K. H. L. A. L. S. M. D. James C. Corbett, "Spanner: Google's Globally-Distributed Database," *OSDI'12: Tenth Symposium on Operating System Design and Implementation,* 2012.

[16] J. W. a. E. E. Ian Robinsson, Graph Databases, O´Reilly Media, 2013.

Appendix

```
//user_rating document, "bucket"
{
  "_id" : ObjectId("535fac50cb01470e4c3407cf"),
  "USER_ID" : 6,
  "ratings" : [{
      "_id" : ObjectId("535fd175cb01470e4c6a1415"),
      "MOVIE_ID" : 1,
      "DATE" : ISODate("2004-01-20T23:00:00Z"),
      "SCORE" : 3
    }, {
      "_id" : ObjectId("535fd4b5cb01470e4c9d48b2"),
      "MOVIE_ID" : 761,
      "DATE" : ISODate("2005-05-19T22:00:00Z"),
      "SCORE" : 4
    }
}
```

Figure 4: data model of user_rating(MongoDB, bucket)

```
// user document
{
  "_id" : 6
}
```

Figure5: User document in MongoDB

```
//movie document
{
  "_id" : 1,
  "title" : "example movie",
  "year" : 2014
}
```

Figure 6: Movie odocument (MongDB)

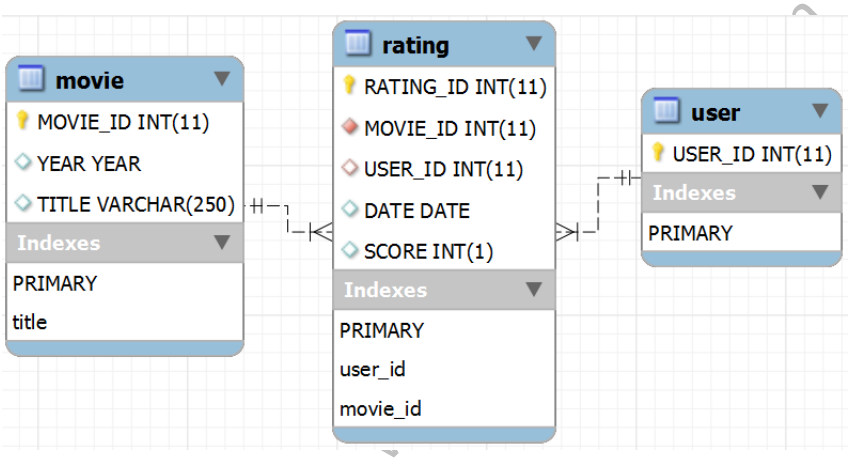

Figure7: ER-Diagram of the database in MySQL