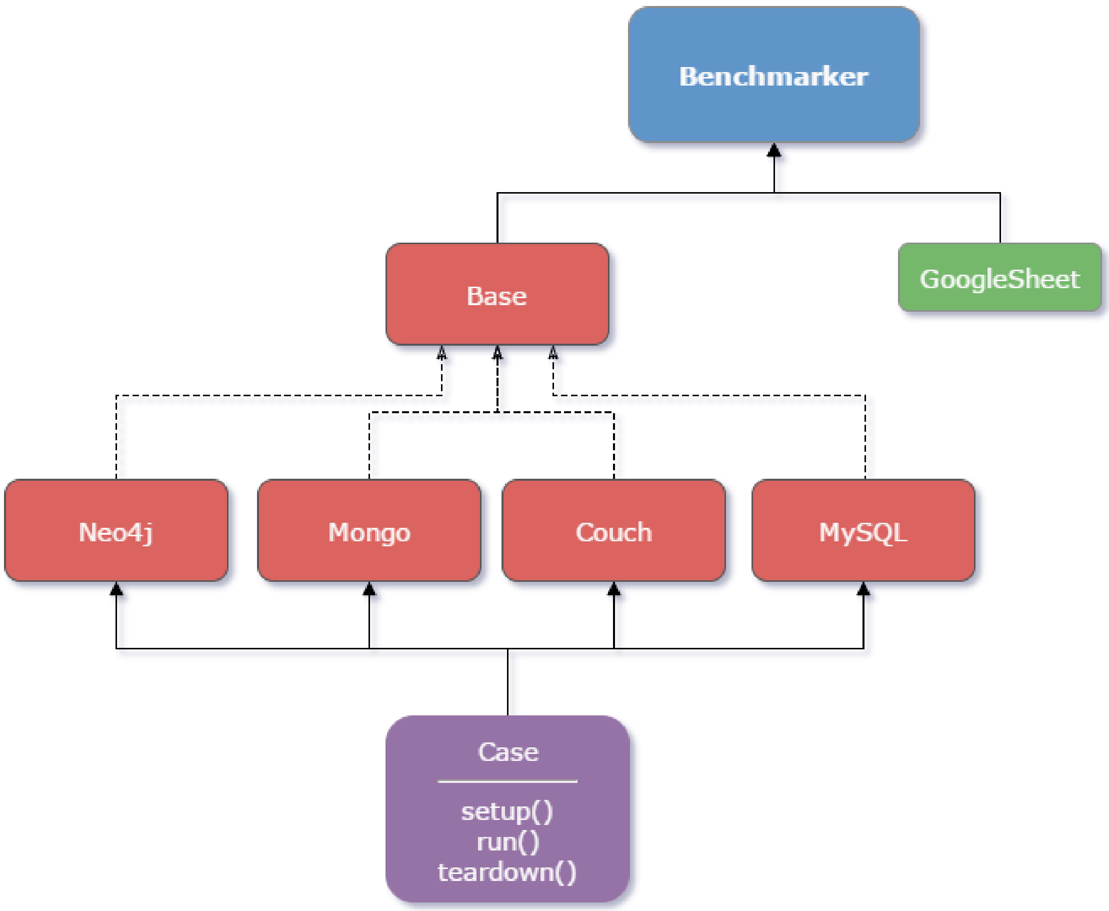
CHALMERS
UNIVERSITY OF TECHNOLOGY



Comparison of Database Management Systems

A comparison of common Database Management Systems

Bachelor's thesis in Computer Science and Engineering

Simon Arneson, Joakim Berntsson, Victor Larsson,
Simon Larsson Takman, Erik Nordmark, Pedram Talebi

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Bachelor's thesis 2016:DATX02-16-16

Comparison of Database Management Systems

A comparison of common Database Management Systems

Simon Arneson, Joakim Berntsson, Victor Larsson,
Simon Larsson Takman, Erik Nordmark, Pedram Talebi

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Comparison of Database Management Systems
A comparison of common Database Management Systems

SIMON ARNESON
JOAKIM BERNTSSON
VICTOR LARSSON
SIMON LARSSON TAKMAN
ERIK NORDMARK
PEDRAM TALEBI

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

© SIMON ARNESON, 2016.
© JOAKIM BERNTSSON, 2016.
© VICTOR LARSSON, 2016.
© SIMON LARSSON TAKMAN, 2016.
© ERIK NORDMARK, 2016.
© PEDRAM TALEBI, 2016.

Supervisor: Morten Fjeld, Department of Applied Information Technology
Examiner: Niklas Broberg, Department of Computer Science and Engineering

Bachelor's Thesis 2016:DATX02-16-16
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2016

Abstract

Data can be referred to as objects or events stored on digital media. The evolution of IT solutions has sprung an interest in developing systems to store this type of data more efficiently. The modern method of storing general data is by utilizing a *database*, which is an organized collection of logically related data.

Different types of databases suit various needs, therefore the purpose of this project is to compare the leading database solutions with each other. The databases chosen for this project are: MySQL, Neo4j and MongoDB.

Furthermore we collaborate with two companies in order to acquire hands-on information of how companies work with data. This is done by developing modules for their web applications. The insight and data obtained through these companies is used to create test cases which is then benchmarked on the different database management systems.

The test results indicates that MongoDB has overall the best performance. However, during a few complex test cases MongoDB lacks in performance. Neo4j performs worse on more test cases than MongoDB, and MySQL is by far the most consistent of these three technologies. The outcome is similar to our expectations and confirms parts of the related work that has been researched.

Keywords: database, benchmarking, test cases, company analysis, nosql, dbms, relational

Sammanfattning

Ordet data innefattar flera tolkningar och kan förklaras som antingen objekt eller som händelser lagrade på digitala medier. Utvecklingen av IT-lösningar har främjats av en strävan efter att lagra data så effektivt som möjligt. Idag lagrar man huvudsakligen data genom att använda en databas, vilket i grund och botten är en organiserad samling av logiskt relaterad data.

Eftersom data kommer i många variationer, lämpar sig olika databasteknologier för olika situationer. Syftet med detta projekt är därför att jämföra några av de ledande databaslösningarna. Valet av databaser blev därför följande: MySQL, Neo4j och MongoDB.

I vårt projekt samarbetar vi med två företag för att förvärva praktisk information om hur företag arbetar med data. Detta görs genom att utveckla moduler för deras webbapplikationer. Den kunskap som erhålls genom att analysera företagen används för att skapa verklighetsbaserade testfall som sedan genomförs empiriskt på de olika databassystemen.

Testresultaten visar att MongoDB presterar generellt bäst. Under några komplicerade testfall underpresterade dock MongoDB. Neo4j presterar sämre på fler testfall än både MongoDB och MySQL. Testresultaten visar även att MySQL är den mest konsekventa av de tre teknikerna. Resultaten stämmer överens på flertalet punkter med våra förväntningar och bekräftar delar av det relaterade arbete som vi undersökt.

Nyckelord: databaser, tester, testfall, företagsanalyser, sql, nosql, relationsdatabaser

Definitions

Data	Objects and events stored on digital media
Database	An organized collection of logically related data
Database refactoring	A change to a database schema that improves its design
Entity	A certain type of data in a model
General data	Commonly occurring, non-complex data such as user profiles, products, orders etc.
Information	Data processed to increase the knowledge of the observer
Large-scale applications	Applications handling big data sets and many queries
Model	Generally defined structure of data and its' relations

Names & Abbreviations

Agile	Collection name for iterative methodologies
API	Application Programming Interface
Auto-sharding	Automatically horizontal partitioning of a database
Benchmarking	Something that can be used as a way to judge the quality of similar things
CMS	Content Management System
CPU	Central Processing Unit
CRM	Customer Relationship Management
CRUD	Create Read Update Delete
CSV	Comma-separated value
Cypher	Neo4j's query language which is based upon SQL
Daemon process	A process that runs in the background and performs a specified operation at predefined times or in response to certain events
Database	Storage solution for data
Database Schema	Schematic describing constraints to be applied on a database model
DBMS	Database Management System
IoT	Internet of Things
JavaScript	Lightweight, prototype based programming language
JSON	JavaScript Object Notation
Map reduce	Query and aggregation technique
NoSQL	Generic name for databases which is not based on SQL
OS	Operating System
Python	Multi-paradigm programming language
RAM	Random Access Memory
RDBMS	Relational Database Management System
Schemaless	The documents stored in the database can have varying sets of fields, with different types for each field
SKU	Stock Keeping Unit
SQL	Structured Query Language
SSD	Solid State Drive
Startup	Newly emerged, fast-growing business
URL	Uniform Resource Locator
VM	Virtual machine

Acknowledgements

We would like to take the opportunity to express our sincere gratitude to the department faculty members Niklas Broberg and Graham Kemp, for the productive discussions during this project. We would also like to express gratitude to our supervisor Morten Fjeld, for all of the support both before and during the project phase.

We also want to thank Marco Schirone, librarian at Chalmers University of Technology, who helped us during the project with questions regarding the reference management system.

We are grateful towards all of the companies that have taken their time do interviews with us and answered our questions. We are especially grateful towards Raceone and Skim for the great collaboration and supplying us with test data.

Finally, we would like to thank Stefan Svensson, our contact person at Skim, for always taking the time to set up weekly meetings and always being eager to give us feedback.

Contents

List of Figures

List of Tables

1 Introduction

Data is a concept with many definitions where books, images, and videos can all contain different forms of data. Data refers to objects or events stored on digital media [2], and the evolution of IT solutions has created a large interest in managing data for the web. The modern way of storing general data, such as texts and numbers, is within a database. Databases are divided into different branches which are suitable for different types and amounts of data. The storage, security and accessibility of data requires a great deal of attention and competence. These requirements can be fulfilled by some companies, but it can be a difficult task for small companies due to the lack of resources.

While the subject of databases and data is growing in complexity, more work has to be done to compile the information to a comprehensible amount. Different companies needs different approaches to store their particular data, especially in order to avoid reworking their entire model in a near future.

1.1 Purpose

The focus of this project is to compare and analyze the performance of different database management systems, DBMS's. Two companies are chosen as collaborators in order to understand what type of data that is stored and how it is used.

The insight and data derived from the companies are used to create test cases of data usage and analyze these on the most used database technologies. The result is meant to be easily understandable and help in the selection of a database strategy.

1.2 Problem definitions

There are several obstacles when benchmarking and comparing DBMS's, the first is the selection of technologies to analyze. The chosen technologies have to be widely used and represent the majority of DBMS's. There are numerous database categories containing different variations of database technologies, therefore the selection has to be done with great care.

Another aspect to consider is the hardware to host the systems on. Some systems are meant to run on certain hardware and by choosing the same type of machine for all DBMS's, some issues may be solved but new might arise.

The next problem is the libraries chosen to communicate with the databases. By using different networking protocols, the results of the benchmark can be misleading.

The final problem is related to the human factor, where even if the problems above are solved, the implementations of clients using the technologies and libraries may be done insufficiently or incorrectly.

1.3 Scope & Delimitations

Three types of databases will be considered during this project. In each of these types, one implementation will be chosen which will represent the whole category. The chosen implementation can also be used in multiple ways, which limits this project to researching a few of the implementations and evaluating the most promising version.

A substantial area that will not be considered in this research, is different sets of hardware. Databases can behave differently on different hardware, however, to include this would make this project too broad. Therefore we will choose a single reference machine to be used by all databases.

In the beginning of the project CouchDB was included as a candidate to be compared. However, it was later excluded due to our priorities.

2

Theory & Tools

The theory behind this project is focused on the implementations of the chosen database management systems and the technologies they are based upon.

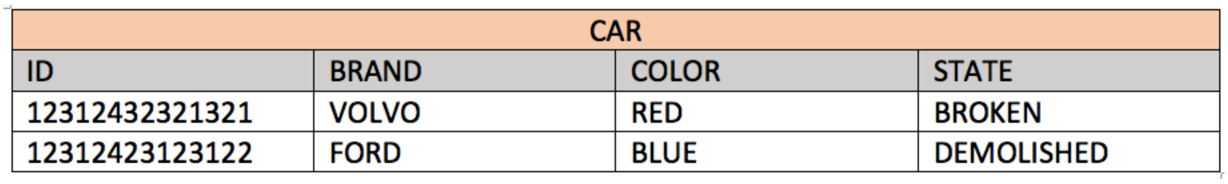
2.1 Types of Databases

A database can be defined as a collection of logically related data [2]. There are several types of databases and two widely used technologies are: *relational* and not only Structured Query Language, *NoSQL*.

2.1.1 Relational

Since the first commercial implementation of the relational database was released by Oracle in 1979, the choice of database solutions within companies has been dominated by the relational database model [3].

A relational database organizes data items into tables, called entities, where each column in the table represents an attribute of that entity. Every row in the table represents one data item, for example one product, and each row must have an unique key. An example of a table can be seen in Figure 2.1. Values in a table can be a reference to a value in a column in another table, and this can be seen as a relationship between columns. [4]



CAR			
ID	BRAND	COLOR	STATE
12312432321321	VOLVO	RED	BROKEN
12312423123122	FORD	BLUE	DEMOLISHED

Figure 2.1: Example of Car data stored in a table.

In relational databases there is a term called constraint, which is a rule that cannot be violated by database users [2].

2.1.2 NoSQL

Many companies run their business and interact with customers primarily through web- and mobile applications, which imposes greater demands on performance and creates new technology requirements. Below is a list of common technological requirements of modern applications [5]:

- Support large numbers of concurrent users (tens of thousands, perhaps millions)
- Deliver highly responsive experiences to a globally distributed base of users
- Always be available – no downtime
- Handle semi- and unstructured data
- Rapidly adapt to changing requirements with frequent updates and new features

Due to the new technological requirements, enterprise technology architecture have to be renewed and adapted to manage real time data in an efficient way. Unprecedented levels of scale, speed, and data variability are some of the new technical challenges which must accommodated. For instance, "over 1 billion people use Facebook every month, and every day there are more than 2.7 billion Likes and over 2.4 billion content items shared with friends" [6]. These circumstances made companies lean towards a new type of database technology, NoSQL. NoSQL is a term for various types of databases, each with their own specific characteristics. Below is a list of common types of NoSQL databases:

- Graph database - Neo4j
- Document database - MongoDB
- Key-Value store - CouchDB

The main difference between relational databases and NoSQL is how their data models are designed. Unlike relational databases where a fixed model is based on static schemas and relationships, the data model in NoSQL is more flexible and does not require predefined schemas. By using the schema-less model, users are able to make modifications in their database in real-time, which fits well with agile development. Due to the flexible data model, modifications in an application leads to fewer interruptions in the development and less database administration time is needed.

2.2 Structured Query Language

A query language can be described as a tool to write predefined functions for extracting data from a database. As early as 1970, Edgar F. Codd published a paper where he explained how to access information in databases without knowing the structure of the information. This was the start of structured query language, SQL. [7]

SQL is the most well known query language, and extensions of SQL is used by both relational database management systems, RDMS's, and NoSQL databases. There are several implementations of the SQL standard and all of them share the *CRUD* cycle. CRUD is an acronym for: Create, Read, Update, and Delete, which are the main operations in the SQL standard.

SQL statements are used to extract information from a database. A statement in SQL is composed out of several keywords, each with its own purpose. A select statement fetches data from the database, it represents the Read operation of the CRUD cycle. The following list consists of the most common keywords used in a select statement to read and structure data [8].

- FROM - Which tables to retrieve data from.
- WHERE - Compares the result and eliminates everything that evaluates as false.
- GROUP BY - Group the result by one or more columns.
- HAVING - Filters groups based on criteria specified in the query.
- ORDER BY - Sorts the result in ascending or descending order.
- DISTINCT - Eliminates all duplicates in the result.

Below is an example of querying the id, price and name of a product stored in the Products table ordered by the price followed by the name [8].

```sql
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price, prod_name;
```

2.3 Database Management Systems

A database management system, DBMS, is a layer between an application and data, which allows the usage of a database approach through CRUD operations. The central positioning of a DBMS is crucial to allow multiple applications and users to interact with the same data, and thus establishing a communication tunnel [2]. This is illustrated in Figure 2.2.

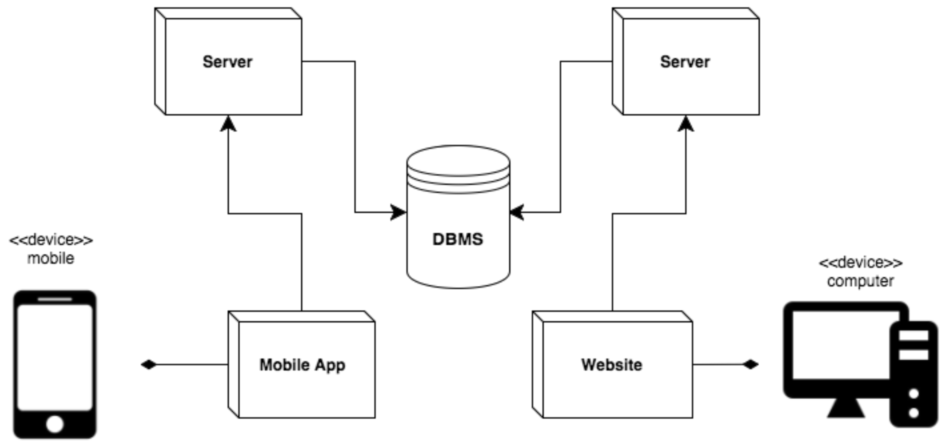


Figure 2.2: Abstracted overview showing the position of a DBMS in an application environment.

Furthermore, a DBMS is responsible for enforcing data integrity, managing concurrency and restoration of data. These three areas are important when expanding the usage of a DBMS to allow more connections and users [2].

2.3.1 Relational - MySQL

MySQL is the second most popular database management system [9]. The MySQL software is released as open source under the GPL licence and it is available on over 20 platforms [10]. It is a relational database management system and uses its own implementation of the SQL standard [11] which differs in some aspects from Transact-SQL, an implementation created by Microsoft and Sybase used in Micosoft SQL Server [12], and PL/SQL, an implementation used in Oracle Database [13]. The main difference between these different implementations is how they handle local variables and procedural processes, such as stored procedures and triggers. [12] [13].

2.3.2 Graph - Neo4j

Graph databases are built upon the idea of graphs, where a graph is a set of vertices and edges which forms nodes and relationships. In our everyday life we can see several similarities to graphs and this common occurrence is why it can be easier to model some things in a graph database. An area where graph databases stands out is during more complex querying of relationships, e.g. for shortest path, minimal spanning tree and other similar algorithms. The most used NoSQL graph database is Neo4j. [14]

There are several differences between Neo4j and RDBMS, where: Neo4j is built on relationships, not entities; Neo4j does not slow down from increasing data sets, since only the number of traversed nodes make a difference; and Neo4j is schema-less, which means it is more flexible to make additions to its model. [14]

According to Ian Robinson et al. [14], the most adopted graph data model is called *labeled property graph model*, where a model is defined as an abstraction or simplification of the real world. This data model is made up of nodes, relationships, properties and labels. Nodes contain properties, in form of key-value pairs, and roles, in form of tags. A relationship has a direction, name, start node and an end node. Like nodes, relationships may also have properties. An example of seven nodes with different relationships can be seen in Figure 2.3.

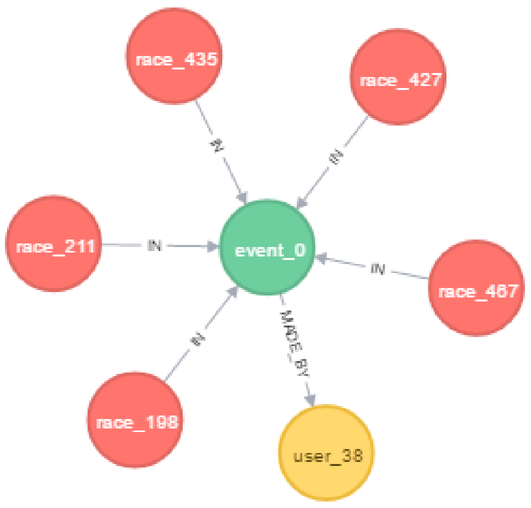


Figure 2.3: Seven nodes with different relationships.

Ian Robinson et al. also mentions Cypher, the standard query language in Neo4j. Cypher is a graphical querying language, with arrows and tags to indicate direction and explanation of a relationship. Cypher's keywords are very similar to the keywords found in SQL and includes the following:

- WHERE - Similar to the SQL WHERE keyword.
- CREATE and CREATE UNIQUE - Used to create nodes and relationships.
- DELETE - Used to delete nodes and relationships.
- MERGE - Updates a node if it exists, otherwise it creates a new node.
- SET - Adds a new property to an existing node or relationship.
- FOREACH - Allows for iteration over a result set.
- UNION - Combines two result sets into one set of results.

Below is an example for querying friends of *Jim* who know each other as well [14].

```
MATCH (a:Person {name: 'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN b, c
```

Validation of a graph model is pretty loosely defined by Ian Robinson et al. and there are no clear strategies to validate a model. One alternative is to begin at the start node and follow its path, while reading the name of each node and relationship along the way out loud. If this creates sensible sentences, then the model is likely to be a good representation. [14]

Common use cases for Neo4j and graph databases in general is for social networking, recommendations and geospatial calculations. A considerable reason for the effectiveness in these areas is the graph and tree related algorithms that can be applied. [14]

2.3.3 Document - MongoDB

MongoDB is the leading NoSQL database [9]. MongoDB is a schema-less document oriented database where data is stored in Binary JSON, a binary-encoded serialization of JSON-like documents [15].

MongoDB is composed of collections, documents, fields and embedded documents in contrast to relational databases, which consists of the traditional tables, rows, columns, and table joins. The Table 2.1 shows the relationship between RDBMS and MongoDB terminology.

RDBMS	**MongoDB**
Database	Database
Table	Collection
Tuple/Row/Record	Document
Column	Field
Table Join	Embedded Documents
Primary Key	*Primary Key (Default key _id provided by MongoDB)*

Table 2.1: Relationship between RDBMS and MongoDB terminology.

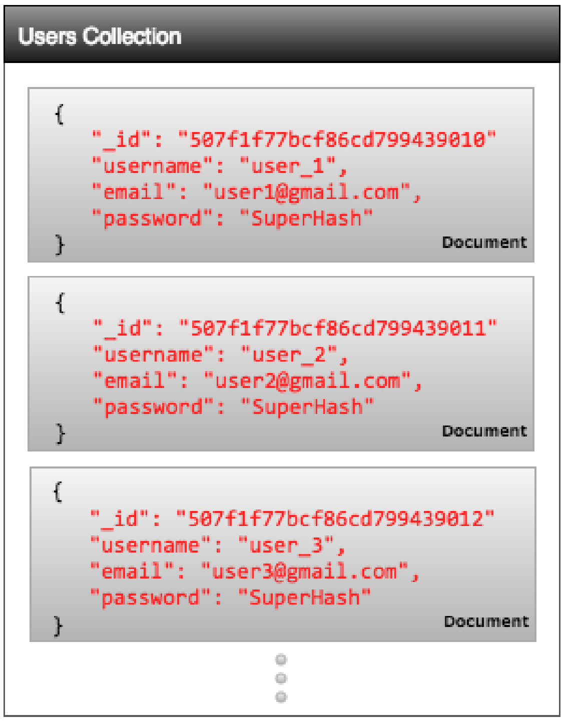
A collection can be seen as a container for the MongoDB documents and since it is a NoSQL database, no predefined schemas are required. MongoDB stores its data as documents and documents have dynamic schemas, which means that documents in the same collection can have different sets of fields. However, every document in MongoDB has a special key, called id, that is unique within a collection. [16]

MongoDB is delivered with a JavaScript shell which allows a user to query a MongoDB database without the use of a library. MongoDB shell uses the javascript dot notation syntax to expose several different types of database operations to the user, such as insert, update, and remove. Each of these functions accept queries formatted as JSON. The MongoDB query language can therefore be considered to be JSON based. [16]

The first of the two code snippets below demonstrates how to add a document with four different fields to the users collection. The second is updating the username field of the document that was added in by the previous snippet. In addition, Figure 2.4 is attached, showing a visual representation of the concepts of documents and collections.

```
db.users.insert_one({
    _id: "507f1f77bcf86cd799439010",
    username: "user_1",
    email: "user1@gmail.com",
    password: "SuperHash"
})
```

```
db.users.update(
    {'_id':'507f1f77bcf86cd799439010'},
    {$set:{'username':'user_1_new'}}
)
```



```
{
    "_id": "507f1f77bcf86cd799439010"
    "username": "user_1",
    "email": "user1@gmail.com",
    "password": "SuperHash"
}
{
    "_id": "507f1f77bcf86cd799439011"
    "username": "user_2",
    "email": "user2@gmail.com",
    "password": "SuperHash"
}
{
    "_id": "507f1f77bcf86cd799439012"
    "username": "user_3",
    "email": "user3@gmail.com",
    "password": "SuperHash"
}
```

Figure 2.4: Visual representation of a collection with documents in MongoDB.

2.3.4 Key-value - CouchDB

CouchDB is a schema-less NoSQL database with a focus on modularization and scalability. CouchDB uses JSON to represent data. Opposed to MongoDB, CouchDB uses a B+ tree storage engine which allows searches, insertions and deletions in logarithmic time. Additionally, CouchDB supports *MapReduce*, which is a query and aggregation technique. [17]

A large number of data structures are linear, since each node can have either zero or one predecessor and successor. Trees are defined differently, where each node can have several successors, or children, yet still only one predecessor, or parent. As stated by Koffman and Wolfgang [1], when searching for an element, linear models have to check each element while sorted trees can limit the continued search to one of the sub-trees. Due to this, linear models have an access time of ordo n while trees can achieve log(n).

B+ trees are also self balancing which means that they try to keep the height of parents' sub trees equal or near equal. While binary trees only allow up to two children, B+ trees supports n number of values for, and children of, a node. Figure 2.5 shows a B tree with children on a per value basis, where each node has multiple values.

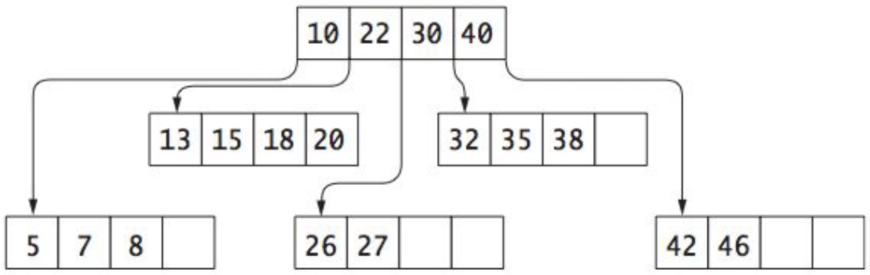


Figure 2.5: Graphical illustration of a B tree [1].

Another aspect of CouchDB is MapReduce. MapReduce consist of two methods, a *map* function and a *reduce* function. The mapping function handles the selection of data while the reducing function aggregates the documents. Both methods return key-value pairs, which are very efficiently queried from a B+ tree. [17]

```
function map(doc) {
    if (doc.type === 'RACE')
        emit(doc.date, 1);
}
function(keys, values) {
    return sum(values);
}
```

Above is an example of a map and reduce function. When enabling the grouping setting, these methods will return all dates and the number of races taking place on that particular date.

When developing with CouchDB, the recommended approach as stated by Anderson et al. [17] is to declare *views* with pre-defined mapping and reducing functions. These views can be accessed with HTTP requests to `http://COUCH_IP:COUCH_PORT/DB_NAME/_design/VIEW_NAME`. This allows for easy GET/POST/DELETE/PUT requests with built in authorization and validation in all languages, since the only requirement is a standard HTTP request library.

Finally, Anderson et al. [17] mentions the deployment of a CouchDB database, which covers scaling through replication, load balancing and clustering. Replication offers synchronization between copies of the same database on different machines. This will enable a distributed network of access points to lower the latency for users all over the world. The access points can also include load balancing, so the machines are used evenly across the network. Load balancing and replication can be used to create clusters and thus induce fail tolerance and a higher performance cap.

3 Related Work

In order to achieve a broader understanding and obtain a glance of what to expect, the group studied some related works of database analysis and evaluated the most promising study below.

3.1 Data Management in Cloud Environments by Katarina Grolinger et al.

The project has three major focus areas. The first area concerns getting a general perspective of the database management domain. This is done by summarizing and categorizing leading NoSQL and NewSQL solutions.

Another focus area of the research is analysing the characterizing differences between the selected data store solutions. The comparison is done in order to act as a guideline towards developers and researchers when they are in the position to choose a data store solution.

The final focus area is about taking all the previous work into consideration and present opportunities and challenges that may be relevant for future iterations, involving the same subject matter.

What we found particularly interesting about this paper was the comparison of the different NoSQL databases. The Results are compiled to help practitioners to choose the appropriate data stores for different use cases. [18]

As stated in Section 2.1.2, NoSQL databases can further be sub-classified based on their data models, and this paper follows the classification provided by Hecht and Jablonski [19].

The three major categories are key-value stores, document stores and graph database [18].

3.1.1 Key-value stores

The study claims that since key-value stores like CouchDB are schema-less, they are well suited to store distributed data, but are not suitable for scenarios requiring relations or structures. It is also stated that key-value stores are appropriate when applications access a set of data using a unique key. Three typical examples are raised in the study where a key value database suits well: storing web session information; user profiles and configurations; and shopping cart data. These use cases suit CouchDB well since the data is accessed through user identification and never queried based on the data content.

Furthermore, the study also found that key-value databases are not the best solution when it comes to highly interconnected data, due to all the relationships that needs to be explicitly handled in the client applications instead. [18]

3.1.2 Document stores

As mentioned in Section 2.3.3 MongoDB store its data as documents. Therefore, applications dealing with data that can be easily interpreted as documents, such as blogging platforms and CMS, are well suited for MongoDB.

Furthermore, it is also stated that a second use case for document data stores is applications storing items of related nature, but with different structure. Consider, for example, where the document data stores are used to log events from enterprise system. The events are represented as documents and events from different sources log different information. However, since the documents use dynamic schemas, changing log formats does not need any database refactoring. Same task for relational database would be tedious, in which a new table needs to be created for each new format or new columns needs to be added to existing tables. [18]

3.1.3 Graph databases

Graph database is the final category that is mentioned. The study found that graph databases, such as Neo4j, is typically used in location-based services, recommendation engines, and complex network-based applications including social, information, technological, and biological networks. A use case that is mentioned in the study is that user location history data, which is used to generate patterns that associate people with their frequently visited places, could be efficiently stored and queried with Neo4j.

Closely related data and complex queries similar to multiple joins in relational databases are also scenarios were graph database is suitable according to the study. [18]

3.1.4 Conclusion of the Study

The study of Katarina Grolinger et al. concludes that NoSQL databases are especially appropriate as alternatives to traditional relational databases in areas where huge volume of data is going to be handled.

In addition, their study claims that a relational database is appropriate in scenarios where the structure is known in advance and unlikely to change.

Finally, the researchers of the study claim that it is essential to consider the investments already made in tools and personnel training prior to selecting the most appropriate database. [18]

4 Approach

This project was divided into four sections: pre-study, implementation, testing and compilation of results. The pre-study consisted of: researching the database technologies; conduction interviews with and developing applications for companies; and analyzing the type of data being used by companies.

Creating test cases was done by discussing the observations of the different companies. Then a benchmarking suite was built with all test cases represented. Finally the cases were run in an interference free environment.

4.1 Market Analysis Company

We collaborated with the company Skim, which was founded in 1979 and has more than 100 employees.

Skim is an international market analysis company which offers pricing and portfolio management to other companies [20]. They majority of Skim's data consist out of images from their clients products. These images require a lot of processing and matching to be useful in an analysis.

To facilitate for their employees they have a small group of developers working with creating an internal customer relationship management, CRM, tool to solve these problems. The CRM platform is a cloud based web portal and has several modules to simplify the communication between Skim and its clients.

4.1.1 System Description

Our task was to develop an *image uploader* module, which will be part of their CRM platform. The *image uploader* is a convenient way for Skim to define rules and requirements for the images sent by Skim's clients. By helping the clients fulfill the rules set by Skim, time and effort can be saved for both parts, as well as communicational improvements between them. The main purpose of our module is to upload images, and then match them to a corresponding stock keeping unit, SKU. A SKU contains multiple attributes, which together defines a product. The image uploader consist of the views: upload, quality assurance, match and confirm.

The finished application is shown in Figure A.1 and A.2.

4.1.2 Suggested Software Models

Figure 4.1, 4.2, and 4.3 are the suggested models for this domain. The models follow the best practices based on external resources, and was based on the use cases that we received from Skim [2] [21]. The developed models was used throughout the development, as well as later in our database tests.

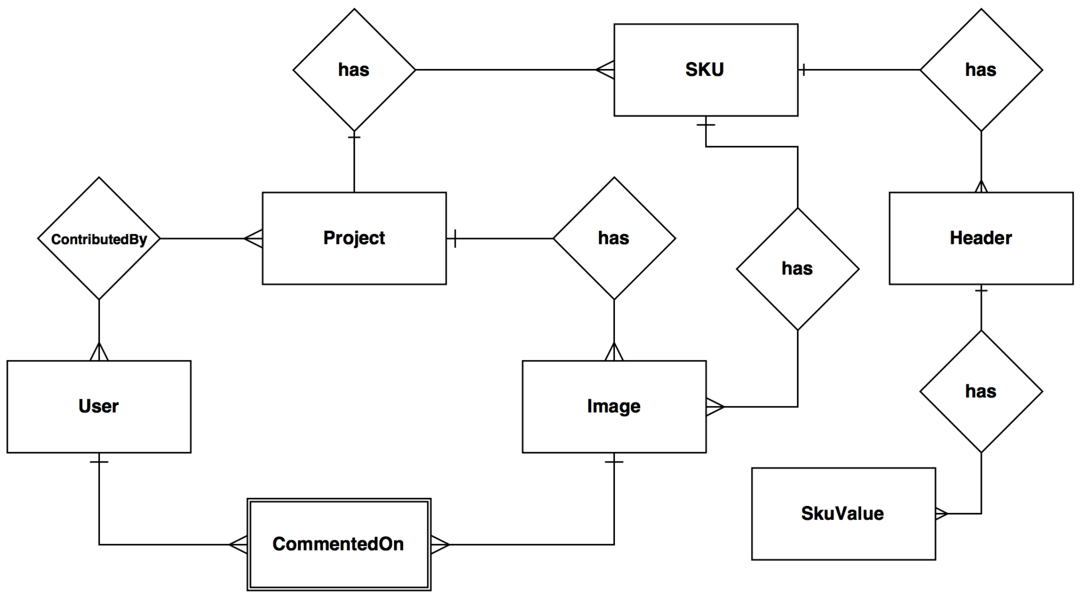


Figure 4.1: Modelling of Skim in MySQL.

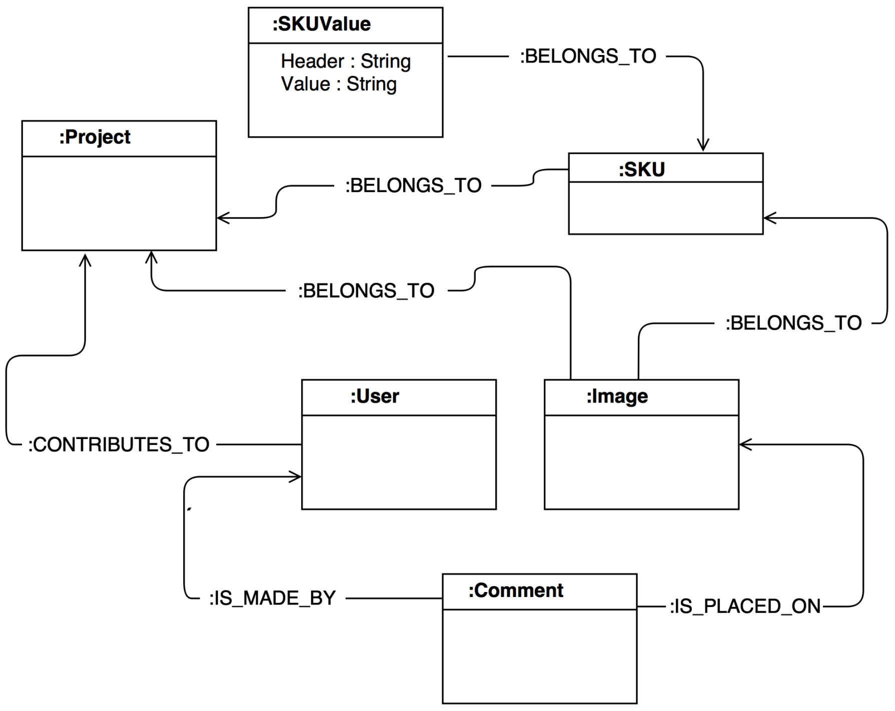


Figure 4.2: Modelling of Skim in Neo4j.

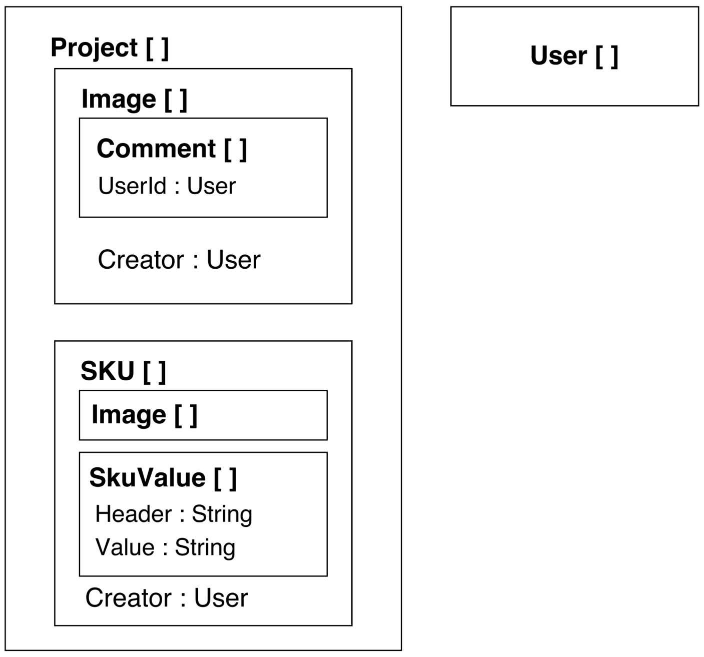


Figure 4.3: Modelling of Skim in MongoDB and CouchDB.

4.1.3 Analysis of Application Usage

The upload phase is responsible for uploading SKU-lists and images. SKUs can be created either by adding them manually, or uploading a CSV-file. The headers for a SKU-list can be different between projects, which lead to a more complex model than initially planned. By introducing headers and values for each SKU, i.e each row value specifies its value with the header, which will give some redundancy. Several image formats can be uploaded to the system and the meta data is extracted and stored in the model before saving the file on the file system. A URL for the image is also stored in the image model.

After uploading the necessary entities, Skim needs to assure that the images upholds the requirements and rules set for the project. At this stage it is possible to comment on images, mark images as *invalid* and delete images. Deleted images is stored in a separate collection in the project. The rules for a project is defined as simple attributes in the project.

To ensure the validity of images, they need to be matched to a SKU. At this point, only images that are *valid* will be shown. When an image is matched, it is removed from the general image list of the project and stored in a list on the SKU. This means that the images are stored in three different lists: the trash list, the image list or within a SKU.

Finally, a confirmation stage is presented. This phase will allow you to get an overview of all SKUs with the images matched to them. When a SKU is deemed *done*, it can be marked as approved and will not be shown in the system anymore.

4.1.4 Suggested Test Cases

The suggested application usage led to the creation of test cases. Our purposed cases are shown in Table 4.1.

Test Cases	
Name	**Description**
Pair Image with SKU	Pairing, or matching, an image with a SKU by removing the image from the project's image list and inserting it into a SKU. This will be a common occurrence in the portal.
Fetch Users	This case will make a request for all users in the database. Fetching users is a typical case for CRM systems.
Add rows to SKU	Adding rows to a SKU-list will take a predefined SKU row and add that to a project. This will occur frequently in Skim's portal and can be problematic with the dynamic SKU headers.
Fetch SKU	Fetching a SKU includes querying all values and headers associated with the SKU. This will be done in several parts of the portal.
Comment on image	Commenting on an image will require: the image id; the id of the commenting user; and the message text as parameters. In Skim's portal this will be their main feature for communicating with clients regarding the image quality.
Fetch a user's comments	Fetching all the comments made by specific user will take a user id as a parameter. This does not occur in Skim's portal at the moment, but could for example be used in a *My Profile* view where a user could keep track of all current conversations.

Table 4.1: Test cases for Skim

4.1.5 Suggested Data Collection

We purpose the amount of test data specified in Table 4.2.

Entity	**Amount**
User	100
Project	10
Collaborator	10
Project Image	100
SKU	20
SKU Value	15
SKU Image	2
Image Comment	5

Table 4.2: Skim entities and their suggested amount.

As indicated by Table 4.2, each project will have 10 collaborators, 100 images, and 20 SKUs. All SKUs have 2 images and 15 values each, and all images will have 5 comments.

4.2 Race Sharing Solution

Raceone is a startup company founded and located in Gothenburg. Their product is a mobile application for sharing and following different sport races live [22]. The participant registers on a race, and during the race the application will send coordinates to a backend, which will analyze the movements. Positional updates will be continuously sent to followers in real time. The coordinates and other information will be saved in their database to enable analysis after a race's completion.

4.2.1 System Description

Our task was to develop Raceone's platform for organizers of various sports race events. Organizers are able to register and handle their events in a simple and convenient way. The application consists of different steps for the organizer to go through, in order to provide sufficient amount of information to set up a new race. Below is a list of the main parts of the application:

- Race registration
- Map registration: Upload, Optimize, Start/Goal and POI's
- Confirmation

The finished application is shown in Figure A.3 and A.4.

4.2.2 Suggested Software Models

Figure 4.4, 4.5, and 4.6 are the suggested software models for this domain. The models follow the best practices based on external resources [2] [21]. The developed models was used throughout the development, as well as later in our database tests.

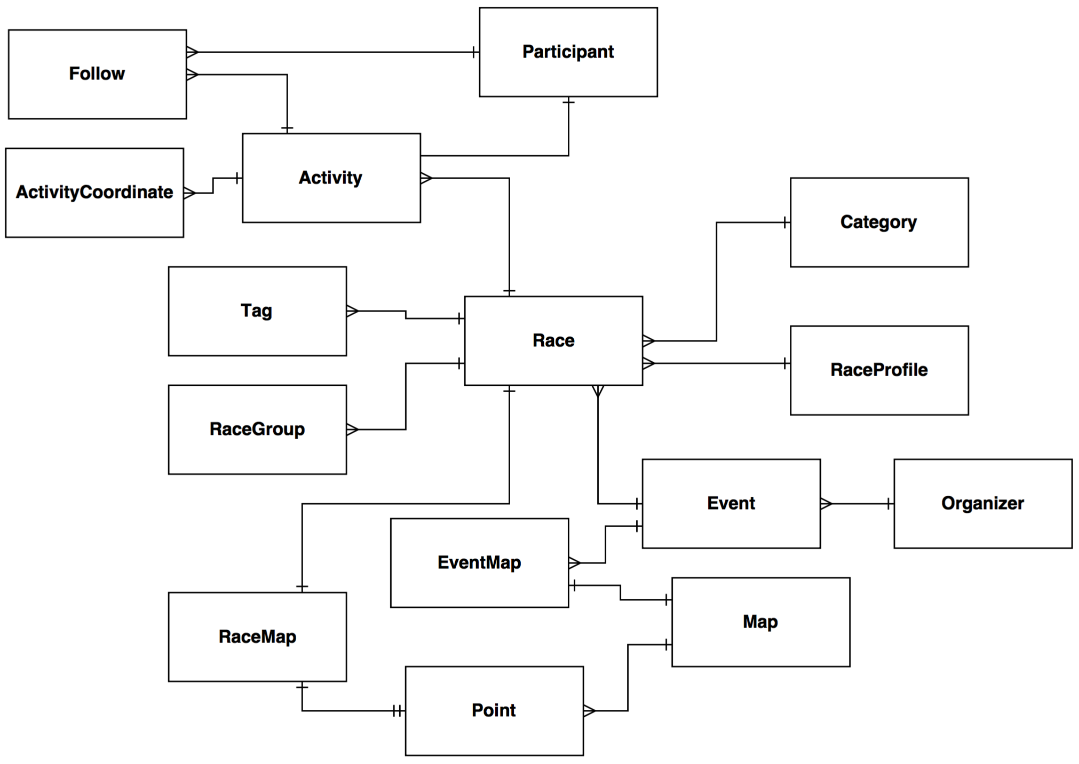


Figure 4.4: Modelling of Raceone in MySQL.

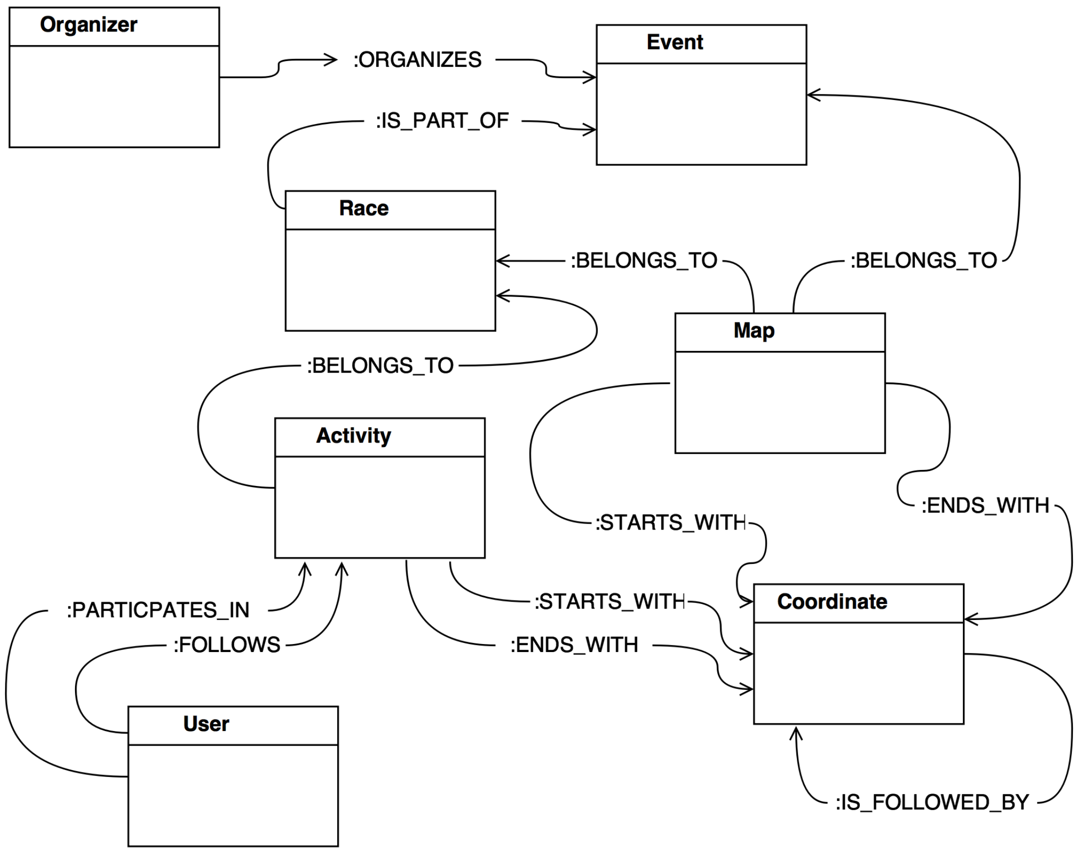


Figure 4.5: Modelling of Raceone in Neo4j.

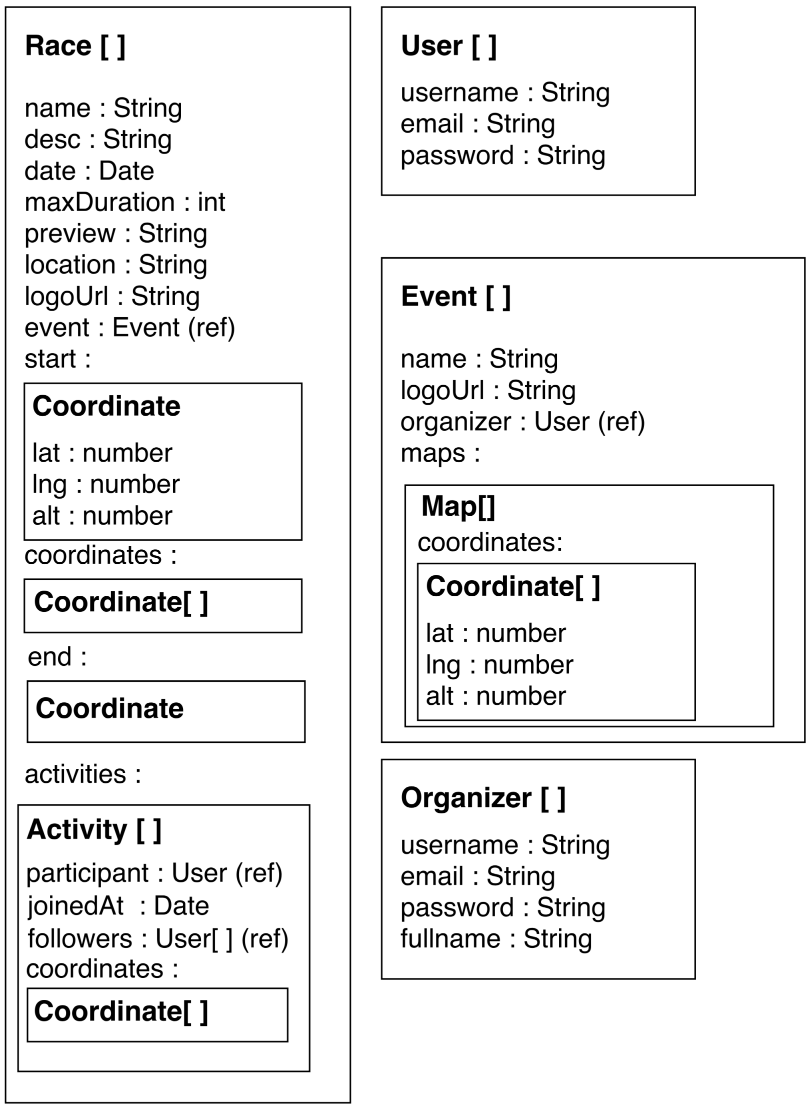


Figure 4.6: Modelling of Raceone in MongoDB and CouchDB.

4.2.3 Analysis of Application Usage

The model and application contains the following main components:

- Organizer
- Event
- Race
- Map

The enclosing layer is the user or more specifically the organizer. The organizer entity holds information such as name, email and password, but primarily it holds the different events. Therefore, the first step in the application is to register an organizer.

The organizer can either choose to create a new event or duplicate a previous event. When creating a duplicate, the new event will have a reference to the predecessor and the old event will have a reference to the newly created successor. The duplicate contains the same information as the previous event.

Every event contains one or more races. If the organizer choose to copy a previous event, all the races in that edition will be copied as well. However, since the new copy is a new edition of that event, the races will also become new editions. Just like events, races contains links to predecessor and any future successor. A race contains all race-related information such as name, date, type of race, maximum number of participants and start groups with associated start time. It also holds information about the race-map.

A race-map is made up of several points which consists of longitude, latitude and altitude. The organizer uploads a race-map to the specific race, and then has the opportunity to optimize and adjust the map. The adjustments could for instance be to change different points of interests, POI, for instance start or finish line.

4.2.4 Suggested Test Cases

The suggested application usage and interviews with Raceone has led to the creation of test cases. Our purposed cases are shown in Table 4.3. Some additional test cases, which will not be explained in detail in this report, are shown in Table A.2.

Test Cases	
Name	**Description**
Duplicate event	The duplicate of an event will contain the original event's races and maps, without duplicating the connected participants and followers.
Fetch race	This case will return a race with the properties: the names of the participants' and their number of followers; the map and; the start and end points of the race. This includes some special querying, counting the number of followers, and also fetching many entities, i.e coordinates.
Follow participant	Following a participant will create a relationship between two users, more precisely, between a user and another user's activity.
Fetch participant	This test case consists of two similar queries, finding the top ten participants which are: participating in the most races; and the participant with the most followers. This is used when displaying followers in the Raceone application and is a common occurrence.
Insert coordinates	Inserting coordinates will be done in batches of one hundred and linked to an activity of a race. For Raceone this is done in the backend to avoid writing to the database too often during a race.

Table 4.3: Test cases for Raceone

4.2.5 Suggested Data Collection

We purpose the amount of test data specified in Table 4.4.

Entity	**Amount**
User	100
Organizer	10
Event	10
Race	5
Race Coordinate	100
Activity	10
Activity Coordinate	50

Table 4.4: Raceone entities and their suggested amount.

As indicated by Table 4.4, each event will have 5 races. A race will include a map of 100 coordinates and 10 activities, where each activity has one participant and one follower, and 50 coordinates as the route traveled so far.

4.3 Reference Point

In Section 1.2 we mention the issue of equal conditions for each technology. When libraries, hardware and human factor plays a part, it is complex to get a point of reference. We have created a simple model and two test cases which are very minimal. This is meant to give an indication of the intermediates between the databases and our test cases, since the cases should be close to identically performed by all technologies.

For a reference, we have chosen a very simple data model and amount of test data. The only entity will be a mock entity, called blob, with only one attribute and no relationships. Furthermore, we purpose the amount of data shown in Table 4.5.

Entity	Amount
Blob	1000

Table 4.5: Reference entities and their suggested amount.

The first test case is not linked to any data sets. Its purpose is to fetch nothing, or a minor package, and return it. This should indicate what the response times are for the different databases. The second case is connected to the model, which consists of one entity and no relationships, and will return all entities found. This can be a reference point for the storage strategy of each database technology.

4.4 Development of Tests

Tests have been developed using the programming language Python and the libraries for DBMS communication as mentioned earlier. The logic is derived from the suggested test cases for all companies. We have developed our own test framework since none exist with the capabilities we were looking for. This framework has the responsibility of automating the benchmarking and synchronizing the results to a spreadsheet for later analysis.

During the development of the test cases, we felt that more time was needed than originally planned. Therefore, a choice was made not to include CouchDB and use the additional time for Neo4j, MySQL and MongoDB. The theory and preparations are finished, so CouchDB can easily be added at a later time.

4.4.1 Test Framework

An internal structure was built to modularize the process of writing the test cases, which allowed distribution of work load while keeping consistency between tests.

The first obstacle was writing tests which realistically extracted data from the DBMS, meaning that because test cases can not use the randomness induced by a user selecting something in a real application, this had to be substituted by a process imitating that behaviour.

The following strategies were considered and tested:

1. Creating the entity before extracting it.
2. Finding the first entity in the database.
3. Extracting all entities and a randomly chosen.

The strategy that best suited our test purposes was number (3).

Intuitively, fetching and randomly choosing an entity would take time, which should not be included in the total time of a test, as it was only done to prepare for the test. This was solved by taking inspiration from unit testing, where each test can include *setup*, *run* and *teardown* methods. With this particular structure, the time elapsed could be measured exclusively for the running part of the test case.

Setup creates, fetches and saves entities along with relationships that are necessary to run the test and to recover the initial state when the test is done. The running method performs the bare minimum to accomplish the purpose of the test case and the teardown cleans up any potential residue to keep the database in its initial state.

A test class was then created for each technology with all cases as methods, where a method would return that case object with the three methods mentioned above. The test collections would also include initialization methods for the different companies.

We also implemented data scaling, as a feature for our Benchmarker. This means that we defined a standard data set, defined in section 4.1.5, 4.2.5 and 4.5, and added methods for expanding the data set with a multiplier. The multiplier can be supplied as an argument, with 1 as default, and will change the initialization of data in the test collections.

Finally there is a benchmarking class which is the backbone of the test environment. This class initializes the data and executes the tests one by one while timing each execution. When the tests are done, the results are synchronized to a Google Spreadsheet for analysis.

Figure 4.7 is a graphical illustration of the test framework. The test collections in red, synchronization module in green, case in purple and the Benchmarker in blue.

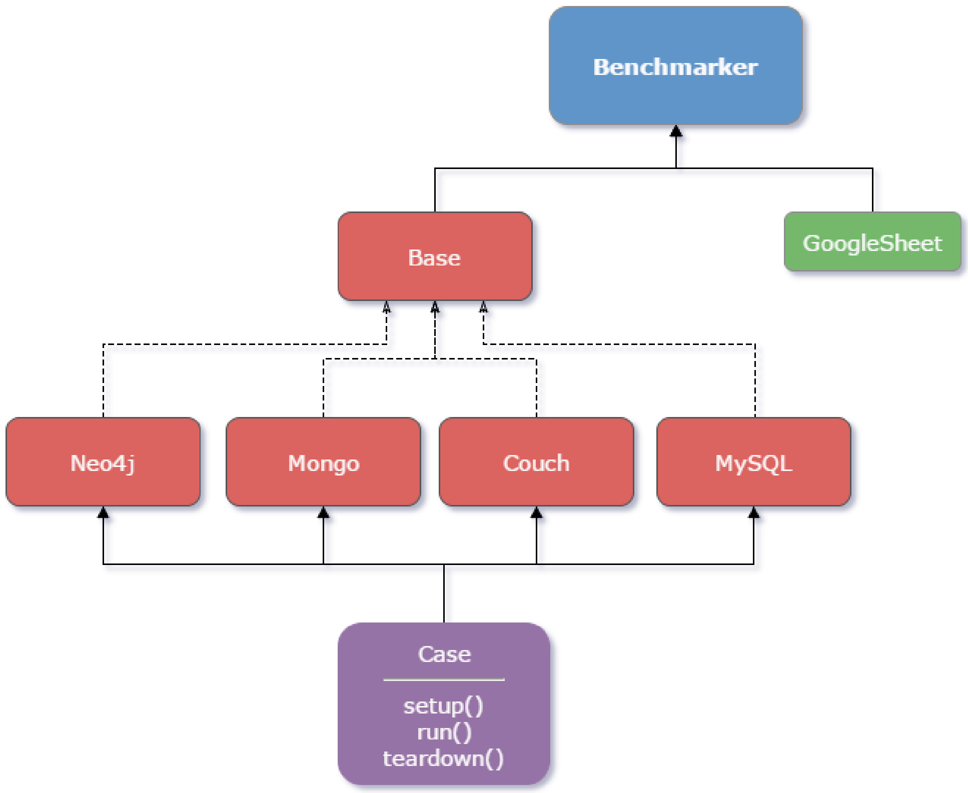


Figure 4.7: Graphical illustration of the test environment.

We used the test framework by running the benchmarking class with arguments for what database and company to test, and an integer for the data scaling.

```
python benchmarking.py neo4j skim 10
python benchmarking.py all all 10
```

4.4.2 MySQL

The tests written for MySQL used the Python library mysql.connector to communicate with the MySQL database. This library utilizes the TCP protocol to create a dedicated connection with a MySQL database. The library connects via TCP/IP sockets to increase the communication performance and SSL to unsure communication security [23].

To connect a MySQL database using this library is rather simple, all needed is this connect statement:

```
cnx = mysql.connector.connect(user="username", password="secret",
    host="ip-address", database="database_name")
```

When a connection has been established you use the context returned by the connect statement to fetch a cursor which you can use to run queries. You can then fetch the results in form of a cursor which you then step through to fetch the data and print it. It is also important to close the cursor when it is no longer in use, since the library only allows one active cursor per connection to the database.

```
cursor = cnx.cursor()
cursor.execute("SELECT id FROM sku")
result = cursor.fetchall()
for row in result:
   print(row)
cursor.close()
```

The most time consuming part of writing these tests were the table creation and initialization of the data.

All queries were written beforehand and tested in an application called DbVisualizer. DbVisualizer is a universal database management tool allowing the user to: connect to a SQL database; run queries against it; and browse the data stored in the database. The application is partly built on open source software and is available to students both as a free version with fewer features as well as through educational discounts on the professional version [24]. This application allowed us to connect to our MySQL server and run queries against it as well as browse the data stored in the database.

4.4.3 Neo4j

To interact with Neo4j through Python, there are several libraries and drivers that can be used. A minimal solution is the Neo4j Python Driver, which supports writing Cypher statements, sending them to the database, executing them, and returning the results [25]. There are several wrappers for this driver, an example is Py2neo, which utilizes the restful API service for Neo4j [26].

Below is a usage example of the driver [25].

```
from neo4j.v1 import GraphDatabase, basic_auth

driver = GraphDatabase.driver("bolt://localhost",
    auth=basic_auth("neo4j", "password"))
session = driver.session()

session.run("CREATE (a:Person {name:'Arthur', title:'King'})")
result = session.run("MATCH (a:Person) WHERE a.name = 'Arthur' RETURN
    a.name AS name, a.title AS title")
for record in result:
    print("%s %s" % (record["title"], record["name"]))
session.close()
```

The first version was written with the wrapper Py2neo, but for performance reasons, we changed to the standalone driver. Py2neo uses Neo4j's restful API service, which was troublesome for us since the libraries for MySQL and MongoDB use the TCP protocol and it performs faster. After the switch to the Neo4j driver, the test cases performed better, and more on par with MySQL and MongoDB.

Another thing to note on the performance of the Neo4j Python library is the usage of parameters. Our first version of the tests were using Python's format string to insert the parameters directly into the query, as shown in the code below.

```
out = self.session.run(
    'START sku=Node(%d) '
    'MATCH (value:SKU_VALUE)-[of:OF]->(sku:SKU) '
    'RETURN value' % inner_self.sku_id
)
```

This was later converted to what the documentation encourages, sending a dictionary with the parameters.

```
out = self.session.run(
    'START sku=Node({sku_id}) '
    'MATCH (value:SKU_VALUE)-[of:OF]->(sku:SKU) '
    'RETURN value',
    dict(sku_id=inner_self.sku_id)
)
```

There are two main benefits of using a dictionary for the parameters. The first one is that queries becomes more efficient thanks to the reusable plans of the query planner. The second benefit is that it reduces the risk of malicious query clauses that derives from inputs that are not properly filtered. These two benefits are stated by Neo Technology [25].

The second benefit is not relevant for our use cases, but it is worth mentioning.

4.4.4 MongoDB

In order to be able to work with MongoDB through Python, a Python driver called PyMongo was utilized. PyMongo is a tool recommended by the official MongoDB website. By using this tool it is possible to connect to the running Mongod instance, which is the primary daemon process of the MongoDB system. The Mongod handles data requests, manages data access, and performs background management operations [27].

```
from pymongo import MongoClient
client = MongoClient()
db = client.test
```

The example above shows how to connect to a test database with help of the MongoClient tool which is imported from PyMongo.

During the process of implementing the data models for the test cases, some flaws in the structure of the models were noticed. Instead of embedding a complete document with corresponding fields inside another document, the embedded document was moved to a separate collection, and only an id reference was embedded in the document. This resulted in more effortless queries and fewer complex embedded documents, and more reads could be done with fewer criteria.

```
db.images.insert_one({
    "name": "image_" + str(nbr),
    "originalName": "original_name",
    "extension": "jpg",
    "encoding": "PNG/SFF",
    "size": 1024,
    "height": 1080,
    "width": 720,
    "verticalDPI": 40,
    "horizontalDPI": 50,
    "bitDepth": 15,
    "createdAt": "2016-03-03",
    "accepted": False,
    "comments": comments
}).inserted_id
```

An example of using references is embedding an image as an object inside another document. The above snippet shows how image documents in our MongoDB test model were added to an image collection. A unique id is added to each image document and this results in less complex query when fields of images are requested.

4.5 Benchmarking

Our benchmarking was run on virtual machines hosted by Digital Ocean, which is a cloud computing service [28]. A test run included all companies and DBMS's, though only one data scale, and ran to completion. There were a total of three runs, one for each data scale, before the compiling of results began.

4.5.1 Virtual Machines

To run our tests we used virtual machines hosted by Digital Ocean. Three different machines were setup with MongoDB, MySQL and Neo4j, with equal specifications, and are specified in Table 4.6.

CPU	Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz (utilizing 2 out of 8 cores)
RAM	2 GB (DIMM)
OS	Ubuntu 14.04.4 x64
SSD	40 GB

Table 4.6: Reference entities and their suggested amount.

In addition to our DBMS machines, we had a test machine, whose sole purpose was to run the test cases. This was done to separate the execution of Python scripts from the DBMS so they would not interfere with each other. All machines were located in the same private network, to avoid latency issues.

4.5.2 Execution of Test Cases

We ran all tests multiple times during the development to analyze preliminary results and evaluate our implementations. After all test collections were done, we ran all tests in batches, one batch for each data scale. As mentioned in Section 4.4.1, we built our test framework to allow data scaling. During our execution we used two different data scales, 1 and 10.

We ran the batches in ascending data scale order, and when the first was completed we analyzed the results to see if there were any issues before continuing with the subsequent batch.

4.5.3 Compiling of Results

The results were analyzed in Google Spreadsheet, where we went through the total, average and peak times. The average times were put in separate spreadsheets for each data scale and we created diagrams for each company. On the horizontal axis of the diagram we had the test cases and execution time average in milliseconds on the vertical axis. We used bar diagrams for the purpose of displaying the results.

5

Results

The results consists of tables of average execution times for the test cases. There are two tables per company, where the first table shows the results for data scale 1, and the second for data scale 10. The tables are used as source data for the following diagrams, which gives an overview of the resulting average times.

In cases where the resulting time is -1 milliseconds, ms, the test was too demanding to write and it was therefore discarded.

5.1 Skim

The results are for all suggested test cases for Skim and each case was run 1000 times.

5.1.1 Tables of Average Times

Table 5.1 and 5.2 are extracts from the spreadsheet of Skim's benchmarking results.

Data Scale 1 [ms]			
Case	MySQL	Neo4j	MongoDB
Pair image and SKU	5,87	5,71	1,33
Fetch users	7,47	27,40	2,12
Add rows to SKU	15,42	7,56	2,81
Fetch SKU	3,78	2,15	1,60
Comment on image	4,89	10,63	1,36
Fetch all user's comments	6,57	18,62	-1,00

Table 5.1: Skim: Average execution times for data scale 1.

Data Scale 10 [ms]			
Case	MySQL	Neo4j	MongoDB
Pair image and SKU	4,12	5,09	1,19
Fetch users	5,71	170,97	9,63
Add rows to SKU	12,18	6,16	2,92
Fetch SKU	2,74	1,98	1,61
Comment on image	3,79	10,89	1,22
Fetch all user's comments	6,22	16,93	-1,00

Table 5.2: Skim: Average execution times for data scale 10.

5.1.2 Diagrams

Figure 5.1 and 5.2 are diagrams created with the data from Table 5.1 and 5.2. Figure 5.2 has been split into two diagrams for clarity.

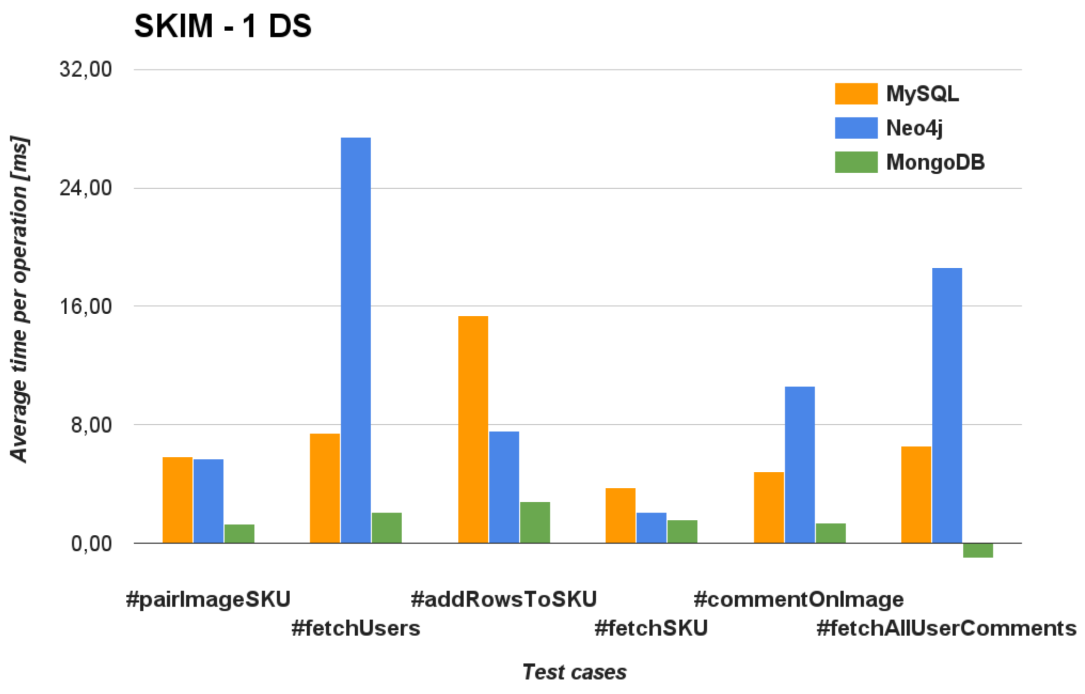
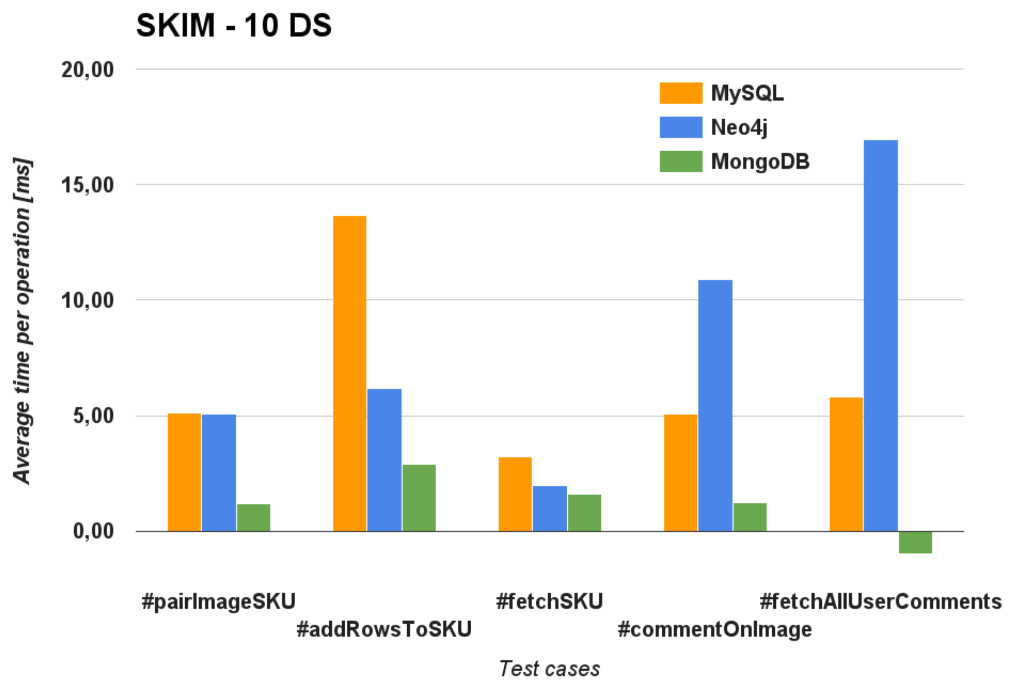


Figure 5.1: Average times for Skim's test cases for data scale 1.



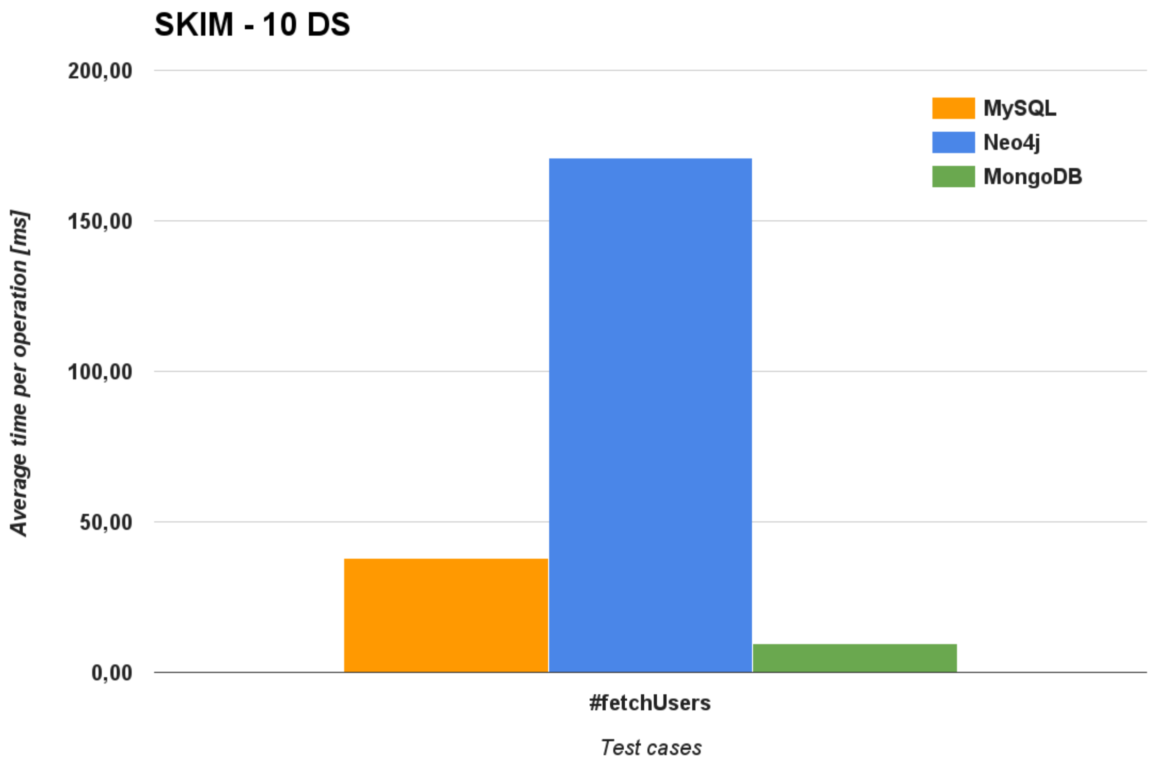


Figure 5.2: Average times for Skim's test cases for data scale 10.

5.2 Raceone

The results are for all suggested test cases for Raceone and each case was run 1000 times.

5.2.1 Tables of Average Times

Table 5.3 and 5.4 are extracts from the spreadsheet of Raceone's benchmarking results.

Data Scale 1 [ms]			
Case	MySQL	Neo4j	MongoDB
Follow	4,37	6,11	1,76
Unfollow	4,37	6,20	1,77
Unparticipate	4,85	9,48	2,10
Fetch participants	2,67	7,44	-1,00
Fetch participants 2	18,51	10,15	-1,00
Fetch coordinates	3,09	8,92	1,43
Remove coordinates	6,56	10,84	2,62
Fetch hot races	3,95	14,31	39,78
Remove race	4,96	-1,00	1,32
Duplicate event	-1,00	-1,00	-1,00
Fetch race	17,30	35,98	3,28
Insert coordinates	85,83	913,06	183,89

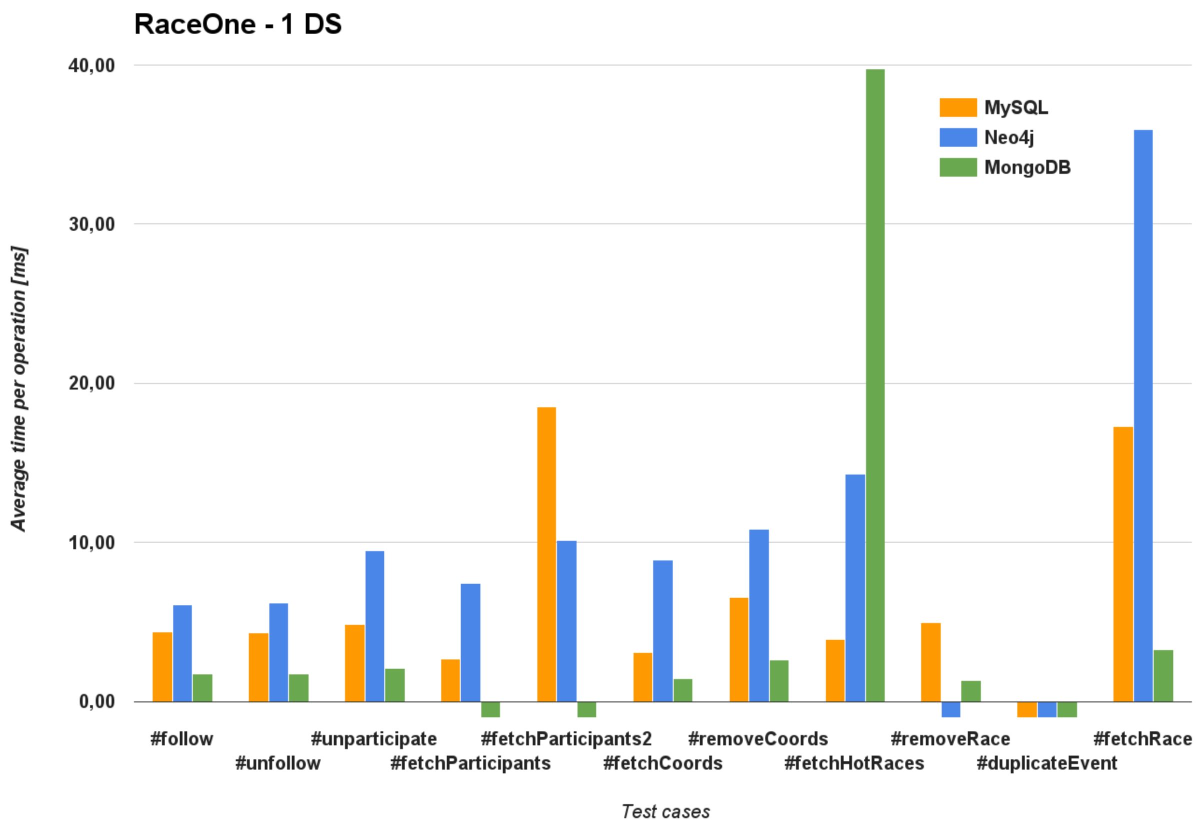
Table 5.3: Raceone: Average execution times for data scale 1.

Data Scale 10 [ms]			
Case	MySQL	Neo4j	MongoDB
Follow	3,71	6,73	2,35
Unfollow	3,64	7,34	2,14
Unparticipate	5,38	14,21	2,54
Fetch participants	8,72	38,24	-1,00
Fetch participants 2	63,28	58,37	-1,00
Fetch coordinates	4,97	33,80	1,83
Remove coordinates	5,80	11,81	2,57
Fetch hot races	18,18	55,62	306,97
Remove race	7,28	-1,00	1,50
Duplicate event	-1,00	-1,00	-1,00
Fetch race	11,97	193,55	3,57
Insert coordinates	145,31	940,03	206,41

Table 5.4: Raceone: Average execution times for data scale 10.

5.2.2 Diagrams

Figure 5.3 and 5.4 are diagrams created with the data from Table 5.3 and 5.4. Both figures have been split into two diagrams each for additional clarity.



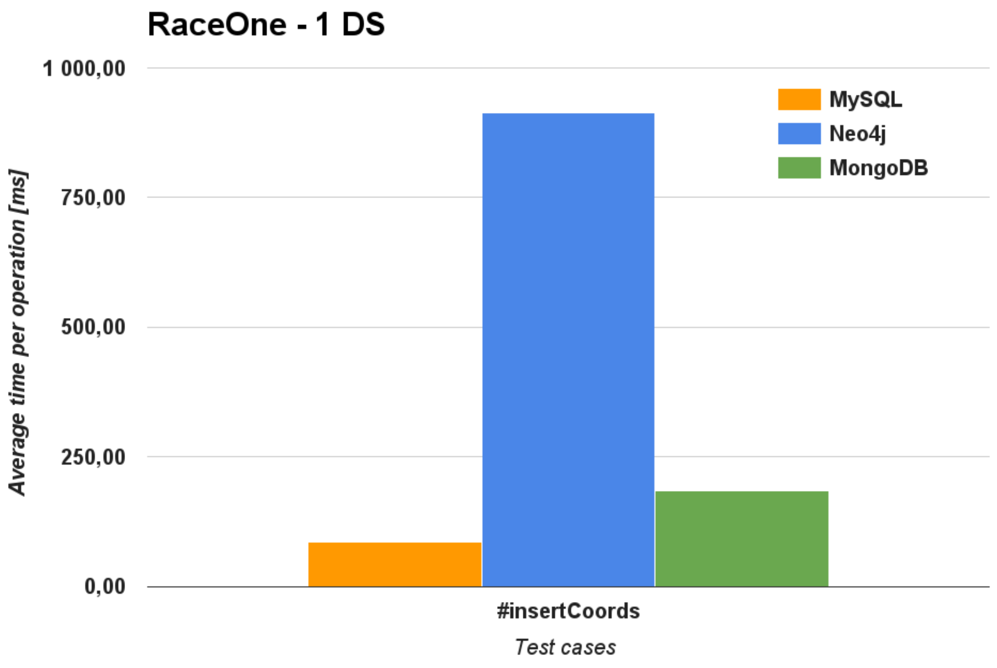
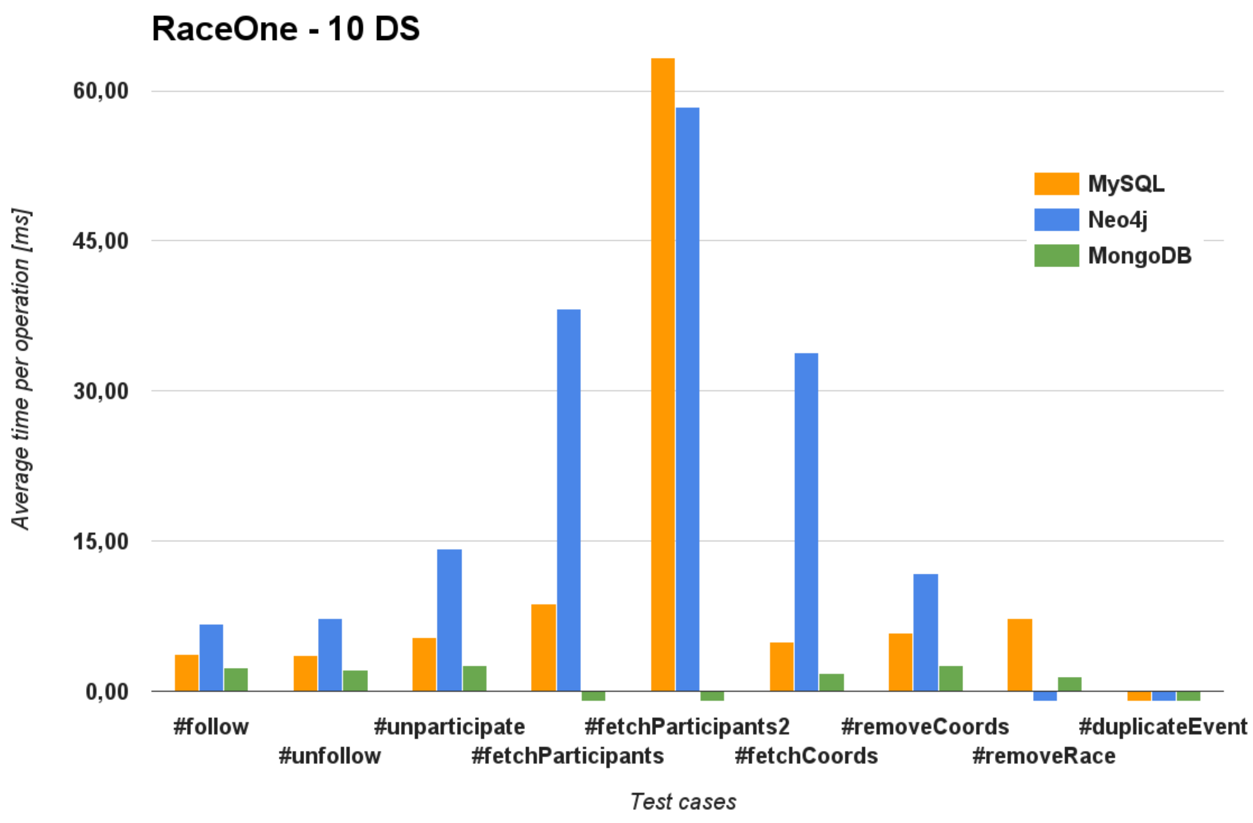


Figure 5.3: Average times for Raceone's test cases for data scale 1.



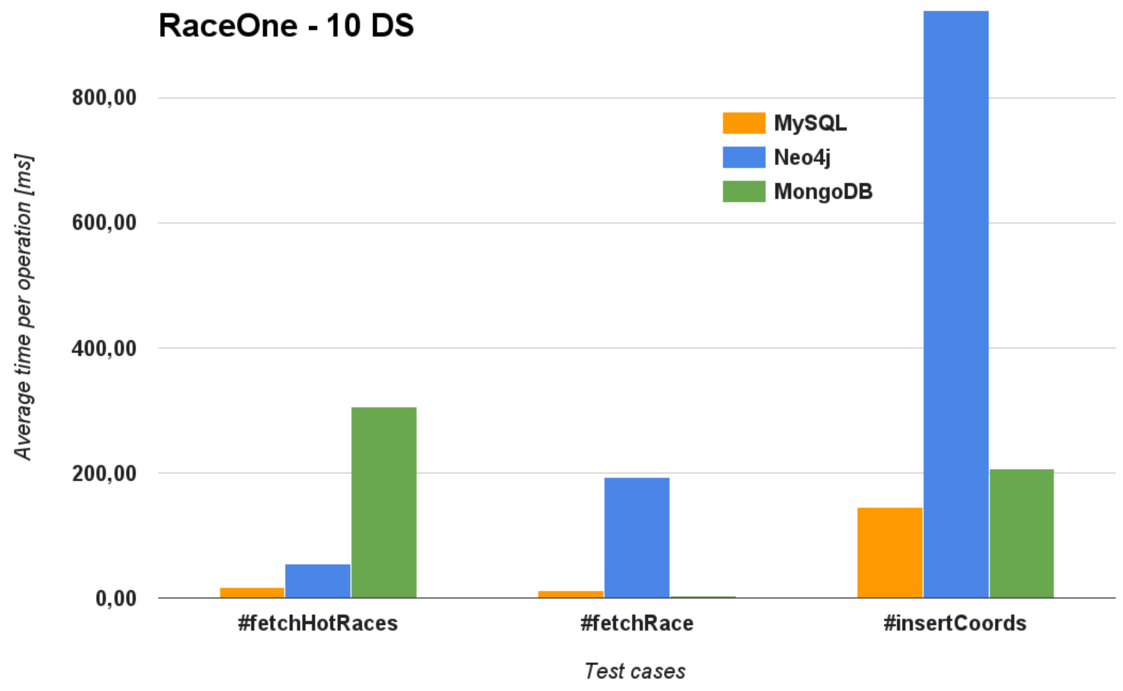


Figure 5.4: Average times for Raceone's test cases for data scale 10.

5.3 Reference

The results are for all suggested test cases for Reference and each case was run 1000 times.

5.3.1 Tables of Average Times

Table 5.5 and 5.6 are extracts from the spreadsheet of Reference's benchmarking results.

Data Scale 1 [ms]			
Case	MySQL	Neo4j	MongoDB
Tiny Get	1,54	1,91	1,49
Small Get	47,04	151,25	11,52

Table 5.5: Reference: Average execution times for data scale 1.

Data Scale 10 [ms]			
Case	MySQL	Neo4j	MongoDB
Tiny Get	1,39	3,32	1,10
Small Get	274,98	1164,35	58,48

Table 5.6: Reference: Average execution times for data scale 10.

5.3.2 Diagrams

Figure 5.5 and 5.6 are diagrams created with the data from Table 5.5 and 5.6.

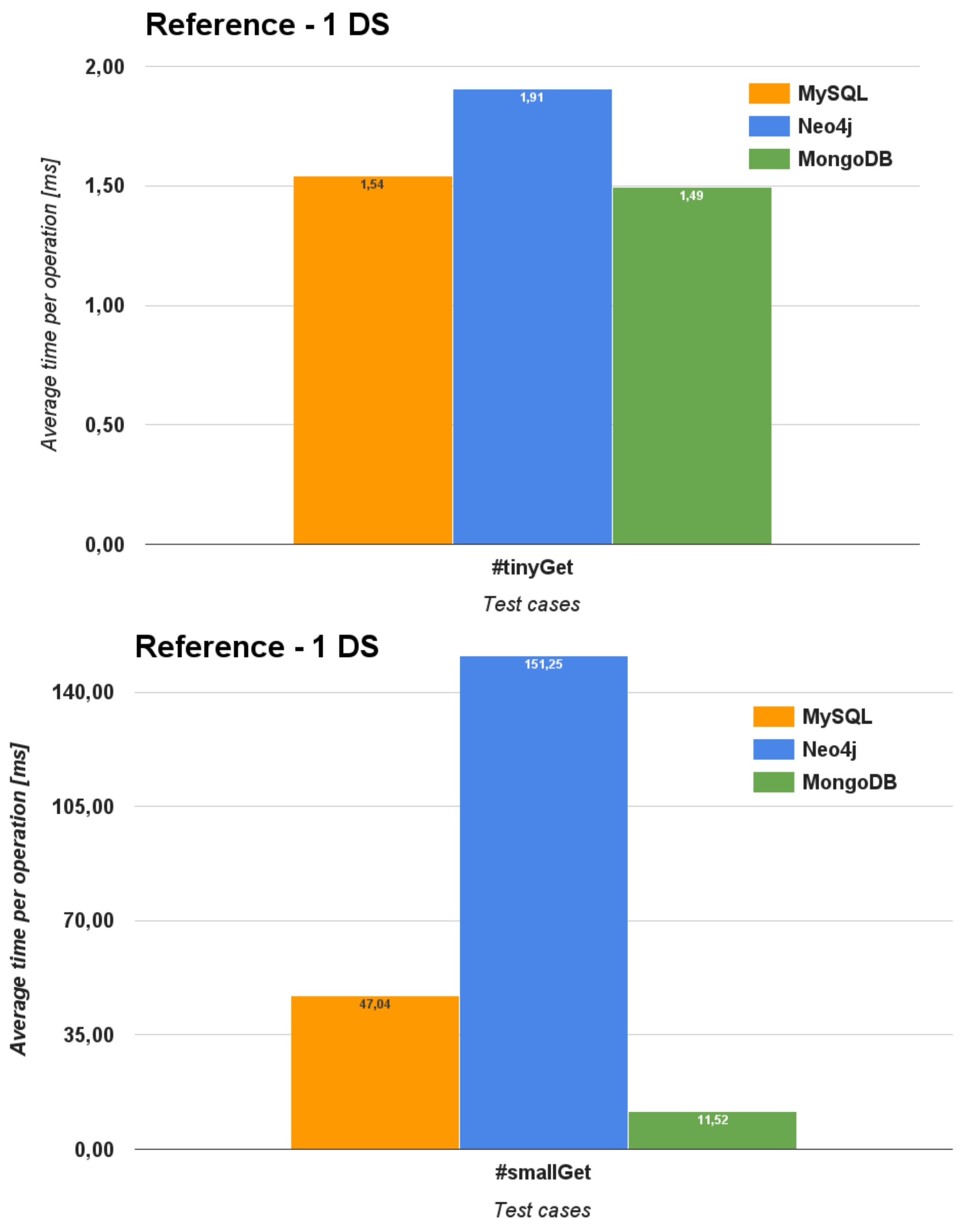
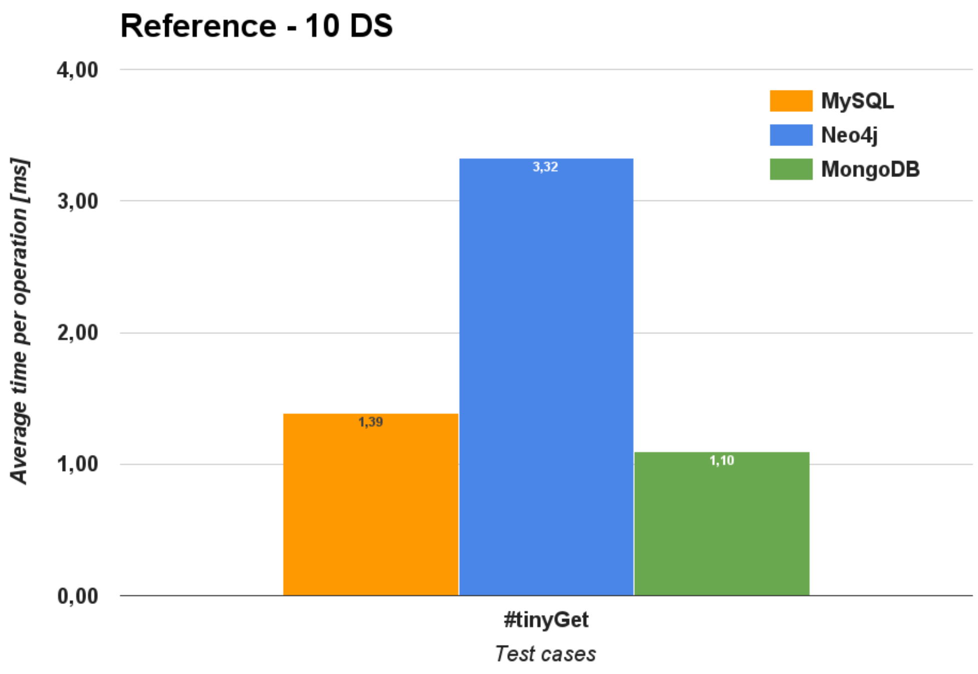


Figure 5.5: Average times for Reference's test cases for data scale 1.



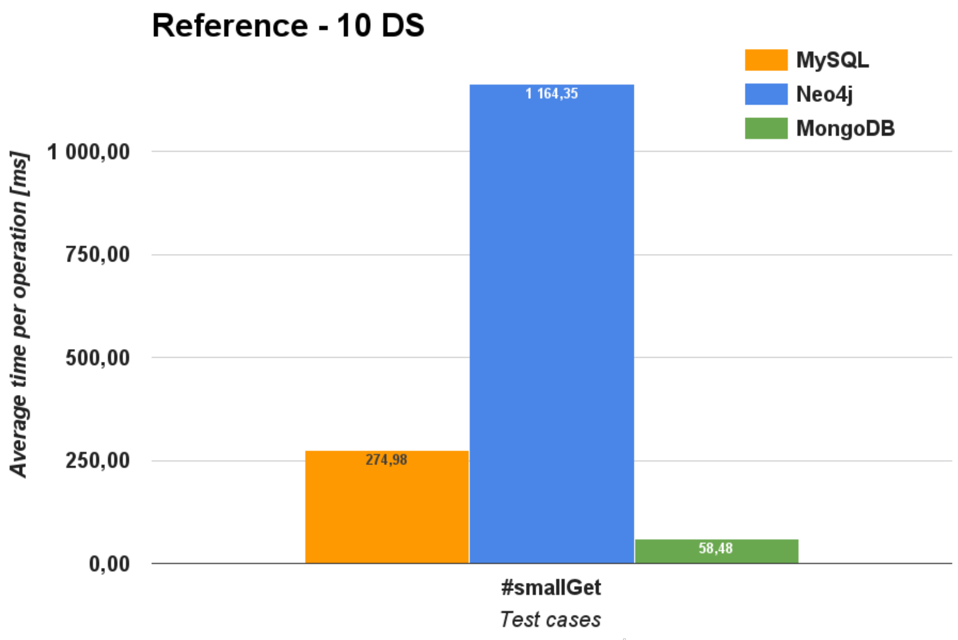


Figure 5.6: Average times for Reference's test cases for data scale 10.

6 Discussion

The purpose of this project is to compare different database management systems. The derived results indicate many interesting factors which will be discussed below, however the majority of the initial expectations comply with the final results.

6.1 Interpretation of Results

The results indicate that the different technologies have individual strengths and weaknesses.

The majority of MongoDB's measurements are close to two milliseconds, which is considerably faster than the results of MySQL and Neo4j. Test cases that differentiates from the low results are complex relationships and joining of data. A summary of MongoDB's results would suggest that it has fast querying overall, with some unimplemented test cases, indicated with a value of -1. The unimplemented test cases were hard, or impossible, to implement considering the chosen model. Current model for MongoDB is optimized for the most common use cases for Skim and Raceone, and is therefore not optimal for some of the other test cases.

The results for Neo4j are generally lacking in performance compared to both MySQL and MongoDB. There is only one test case where Neo4j is the fastest, *fetch participants type 2*. There are some cases, *fetch hot races*, *Add rows to SKU* and *fetch SKU*, where Neo4j is faster than MySQL or MongoDB. However, Neo4j's main weakness throughout the results is when queries are fetching large data sets, where in some cases Neo4j is up to 20 times slower than MongoDB.

The results indicates that MySQL is the most consistent of the three database technologies. While MongoDB and Neo4j fluctuate between test cases, MySQL was more stable.

Considering the scaling of data, MongoDB handles the increase without any major spikes in the results. However, Neo4j slows down considerably on some cases such as *fetch users* and *small get*. This indicates that the performance of Neo4j decreases when scaling up the data size.

6.2 Expectations of Test Cases

Expectations were developed with the help of the related work and theory of this project.

6.2.1 Skim

Table 6.1 and 6.2 contains individual test cases for Skim with their expectation and outcome.

Test Cases	
Name	**Evaluation**
Pair Image with SKU	**Expectation:** The expectation was in favor for MongoDB, since the model for MongoDB has a separate collection for the SKUs. **Outcome:** It was proven that the expectation was correct. MongoDB is in favor due to the model for MongoDB which has SKUs as a separate collection, instead of a nested. The only operation which has to be performed is adding the image ID to the SKU and removing it from the list of image IDs in the project. For MySQL, it is quite similar, aside from one big difference - reference constraints. MySQL will have to find the image and the SKU to perform the update. Finally Neo4j is not based upon constraints, although it needs to find the nodes to be able to connect them, thus the same events as for MySQL will occur.
Fetch Users	**Expectation:** The expected outcome for fetching users was that MySQL and MongoDB will have similar response times. Mainly because they only need to return a single collection while Neo4j have to work with a larger data set, which will be slower even with indexing. **Outcome:** After the test run it turned out that the expected outcome for *Fetch users* was not correct. From the results it is clear that MongoDB has a shorter average execution times than MySQL, the outcome for Neo4j was correct.

Table 6.1: Expectations and their outcome for Skim

Test Cases	
Name	**Evaluation**
Add rows to SKU	**Expectation:** Since the model for MongoDB has a separate collection for SKUs, expectations were that it has the potential of having the best performance for this case. **Outcome:** Outcome of this test case proved that the expectation was correct.
Fetch SKU	**Expectation:** The expectation for this test case was that it will have similar querying times for all technologies, since the models are comparable in this regard. **Outcome:** The results indicated once again that MongoDB performs best, which contradicts the expectation. This may be due to the fact that collections in MongoDB are less heavy to fetch than tables in MySQL, or nodes in Neo4j.
Comment on image	**Expectation:** Regarding the performance for this test case, the expectations were that MySQL will perform the best, this due to the fact that MySQL holds the comments in a separate table and only needs to do an insert into this table to perform the task. In addition, it was also expected that the performance of MongoDB will be very dependent on the implementation of the code running this query. Expectations regarding MongoDB was that if this test was separated into commenting on images within SKUs and commenting on images not connected to a SKU the performance will be acceptable. If the test was not split up and one query is to cover both these cases, the performance will decrease drastically. **Outcome:** Results for this test case was as expected. Important to mention is that the MongoDB test was based on having two different commenting functions as suggested in the expectation.
Fetch a user's comments	**Expectation:** The expectation was that MySQL will be best suited for this task, since the comments are stored in a separate table and each comment contains the user id. MongoDB and Neo4j shared issues with each other since they have to look through every image for comments and then look at the creators of these comments. MongoDB had yet another issue, since it has to search both the Image collection and the SKU collection to find all comments. This is because some of the images are stored within a SKU and some of them are stored in a separate collection. **Outcome:** MySQL performed best as it was expected and from the results it is stated that it was not possible to implement the MongoDB solution for the test case.

Table 6.2: Expectations and their outcome for Skim

6.2.2 Raceone

Table 6.3 and 6.4 contains individual test cases for Raceone with their expectation and outcome.

Test Cases	
Name	**Evaluation**
Duplicate event	**Expectation:** The expected outcome for duplicating an event was that it will be very simple to solve for MongoDB, perform reasonably well in Neo4j and lack performance for MySQL. This is due to all of the nested models this deep copy would have to go through. In MongoDB all of the data connected to a race is stored within a race document, which makes it easy to duplicate. **Outcome:** The expectation can be neither approved nor disapproved, since the test case was not evaluated for any of the technologies.
Fetch race	**Expectation:** Similarly to duplicating events, this case's expected outcome is that it will perform well in MongoDB by returning a whole document. In Neo4j and MySQL the query has to fetch properties from multiple locations and combining them, thus increasing the complexity of the task. **Outcome:** The results indicated that the expectation was correct for all threee technologies.
Follow participant	**Expectation:** The expectation for this test case is that this case will be very fast for all chosen databases, and possibly slightly slower for MySQL, as a result of its referential constraints. **Outcome:** The outcome for this test case confirmed the expectation.

Table 6.3: Expectations and their outcome for Skim

Test Cases	
Name	**Evaluation**
Fetch participant	**Expectation:** MongoDB's weaknesses includes complex sorting and joining queries. Therefore, the expected outcome for this case was that Neo4j and MySQL will be the fastest for this case. **Outcome:** The expectation can not neither be approved nor disapproved, since the implementation of MongoDB's is not completed.
Insert coordinates	**Expectation:** This sort of batch creation should favor MongoDB. **Outcome:** The expectation is partially confirmed, since MongoDB performed well on this test case. However, the technology that performed the best was MySQL, which was not mentioned in the expectation.

Table 6.4: Expectations and their outcome for Skim

6.3 Similarities to Related Work

In Section 3.1.2 it is stated that an appropriate use case for document data stores is for applications storing items of related nature, but with different structure. This could be compared to the test case *add rows to SKU*, from Section 4.1.4, where each SKU has different columns, and amount of columns.

By analyzing the average time per operation for the test case *add rows to SKU* in the diagram, 5.1.1, it is demonstrated that MongoDB and its document model is the fastest of all technologies in this regard. Thus, the results of the test case accords with the study by Katarina Grolinger et al.

The research conducted by Katarina Grolinger et al., which is presented in the related works Section 3.1.3, argues that graph databases performs well when it comes to complex queries. Neo4j performed best of all database systems in the test case *fetch participants* which is an example of such a query.

The same section also states that a common usage of graph databases is when handling location history of a user. The test case *insert coordinates* stores coordinates synced from a race participant's movements along the track. In contradiction to what is argued by Katarina Groliner et al. Neo4j had a very high average time per operation in this test and the other compared database systems performed better.

6.4 Social Aspect

For several years, Chalmers has put a lot of time and effort into its innovative program, Chalmers Ventures. They supply new companies with supervisors and early investments to get them past the initial startup phase. [29]

An issue with the current concept is the neglecting of competence assistance. While the supervisors aid in business and legal aspects, they do not always provide enough competence assistance in the extent that is necessary.

The test cases and results of this project can be used as a tool for companies in their startup phase to steer the solution in the right direction. This in turn can be further developed to be more comprehensive and cover more technological areas for companies.

7
Conclusion

The related work done by Katarina Grolinger et al. in Section 6.3 is mostly confirmed by our results. An aspect in which our results differ from their results is regarding storing location history in a graph database. The reason for this could be insufficient modelling and test implementation during our project. Assuming that our test *inserting coordinates* would execute each insertion by itself instead of doing the insertion in batches, Neo4j would most likely perform better.

For smaller applications with general data, we would recommend the use of MongoDB for its agile compatibility and fast querying. In large-scale applications and solutions where the integrity of data is of greater importance, MySQL or other relational DBMS's would be more compatible. A relational DBMS allows constraints to be added to a model, which in turn leads to a more controlled and defined environment for the data. Our results for MySQL were also consistent and satisfactory in general.

The test results have proven Neo4j to be effective on complex queries with highly coupled data, which was suggested in Section 3.1.3. Further, we have concluded that Neo4j is not suited to fetch large amounts of general data. As mentioned in the interpretation of the results in Section 6.1, the test performance of Neo4j decreases when scaling up the data size. This may have been caused by a lack of correct indexing.

A conclusion which can be made from what is stated above is that different technologies are better suited for different purposes and database models. Therefore, it is an advantage for companies to examine what type of technology is appropriate for their application area. As a final conclusion, we would like to emphasize the importance of evaluating the existing knowledge within a company to avoid mistakes being made because of insufficient expertise, while still having the future in mind.

8 Outlook

Considering the scope and limitations of this project, there are several areas that can be improved or added. At a later stage in the project, during the implementation of the DBMS, the same decision was regarding CouchDB.

Another thing to mention is the resulting measurements for the test cases. Initially, there was plans to include load percentage in addition to execution times, though due to issues such as synchronization it was omitted.

8.1 Additional Companies

Adding more companies with different types of data sets would be very beneficial for continuous work in this area. Social networks would work as an example of closely connected data sets, which is where graph databases are good to utilize. Examples of social networks that were discussed are: Facebook, and Reddit.

Aside from social networking, companies with data for recommendation services or big data would be interesting additions.

8.2 CouchDB

In Section 2.3.4 a short introduction of CouchDB is presented. The research suggests that its strengths in fast access to simple objects and the pre-calculation of views would result in interesting results for the test cases. The preparations are made in the benchmarking model and for the test cases, so to add this technology, minimal additions have to be made to the entirety.

8.3 Analyzing CPU Load

In addition to timing the execution of the tests, it would be beneficial to see the load on the CPU during the execution. If two technologies have similar performance, this knowledge could indicate if one technology is less demanding while being equally fast. The issues that were run into with this feature were that the collection of CPU loads would have to be conducted on the DBMS machines and then sent to the test machine. This would require some logic on the DBMS machines as well, to keep track of the current test case and send the results back to the test machine.

8.4 Displaying of Results

An interesting topic throughout this project has been how to present the results in the most efficient manner. This could be done by developing a simple website with a step-based process to establish what a user needs. The step process could include questions to guide a user through the website.

Bibliography

[1] E. B. Koffman and P. A. T. Wolfgang, *Data Structures: Abstraction and Design Using Java*, 2nd ed. Wiley, 2010.

[2] J. A. Hoffer, V. Ramesh, and H. Topi, *Modern Database Management*, 10, Ed. Pearson, 2011.

[3] Db-engines, "DB-Engines Ranking per database model category," 2014. [Online]. Available: http://db-engines.com/en/ranking_categories

[4] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 2nd ed. McGraw-Hill Companies, 2000.

[5] CouchDB, "Why NoSQL ?" 2016. [Online]. Available: http://www.couchbase.com/nosql-resources/what-is-no-sql

[6] Jay Parikh, "A New Data Center for Iowa | Facebook Newsroom," 2013. [Online]. Available: http://newsroom.fb.com/news/2013/04/a-new-data-center-for-iowa/

[7] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," 1970. [Online]. Available: http://www.acm.org

[8] B. Forta, *Sams teach yourself SQL in 10 minutes*, 4th ed. Pearson, 2013.

[9] Db-engines, "DB-Engines Ranking - popularity ranking of database management systems," 2015. [Online]. Available: http://db-engines.com/en/ranking

[10] Oracle, "MySQL :: MySQL Community Edition," 2016. [Online]. Available: https://www.mysql.com/products/community/

[11] ——, "MySQL :: MySQL 5.7 Reference Manual :: 1.3.1 What is MySQL?" 2016. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html

[12] Microsoft, "Tutorial: Writing Transact-SQL Statements," 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms365303.aspx

[13] S. Moore, "Oracle Database PL/SQL Language Reference, 11g Release 1 (11.1)," 2009. [Online]. Available: http://docs.oracle.com/cd/B28359_01/appdev.111/b28370/overview.htm#

[14] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New opportunities for Connected Data*, 2nd ed. O'Reilly Media, 2015.

[15] MongoDB Inc., "BSON - Binary JSON," 2013. [Online]. Available: http://bsonspec.org/

[16] K. Chodorow, *MongoDB: The Definitive Guide*, 2nd ed. O'Reilly Media, 2013.

[17] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB The Definitive Guide*, 1st ed. O'Reilly Media, 2010.

[18] K. Grolinger, W. a. Higashino, A. Tiwari, and M. A. Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2, no. 1, p. 22, dec 2013. [Online]. Available: http://www.journalofcloudcomputing.com/content/2/1/22

[19] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison," *Nosql database: New era of databases for big data analytics-classification, characteristics and comparison*, vol. 6, no. 4, pp. 1–14, 2013. [Online]. Available: http://arxiv.org/pdf/1307.0191v1.pdf

[20] SKIM, "SKIM | About us." [Online]. Available: http://skimgroup.com/history

[21] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book 2nd Edition*, 2nd ed. Pearson, 2009.

[22] Raceone, "HOME | RaceONE," 2015. [Online]. Available: http://www.raceone.com/

[23] Oracle, "MySQL :: MySQL Connector/Python Developer Guide :: 1 Introduction to MySQL Connector/Python," 2016. [Online]. Available: https://dev.mysql.com/doc/connector-python/en/connector-python-introduction.html

[24] DbVis Software AB, "About Us - DbVis Software," 2016. [Online]. Available: https://www.dbvis.com/company/

[25] Neo Technology, "Drivers," 2016. [Online]. Available: http://neo4j.com/docs/developer-manual/current/#driver-manual-index

[26] N. Small, "Py2neo v3 Handbook." [Online]. Available: http://py2neo.org/v3/

[27] MongoDB Inc., "Python Driver (PyMongo) — Getting Started With MongoDB 3.0.4," 2016. [Online]. Available: https://docs.mongodb.com/getting-started/python/client/

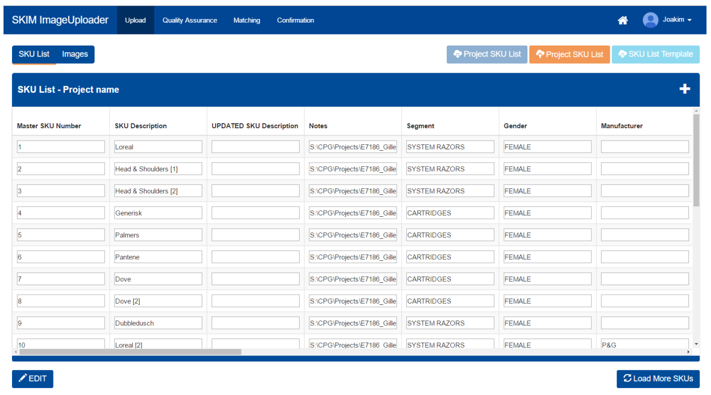
[28] DigitalOcean, "What is Cloud Hosting?" 2016. [Online]. Available: https://www.digitalocean.com/what-is-cloud-hosting/

[29] Chalmers Ventures AB, "About the Programs," 2015. [Online]. Available: http://www.chalmersventures.com/offer

A

Appendix 1

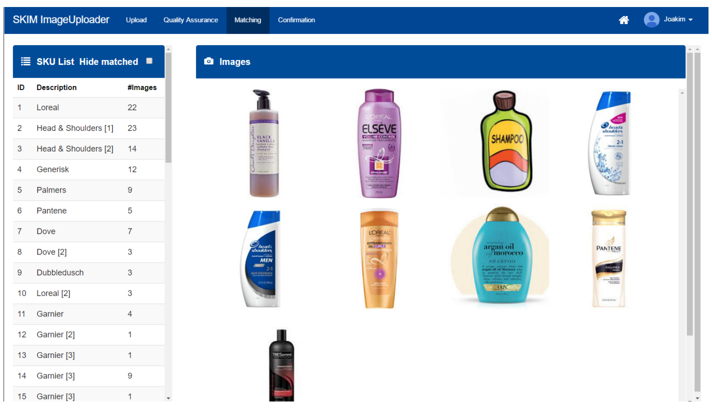
A.1 Screenshots of the company modules



SKU List - Project name

Master SKU Number	SKU Description	UPDATED SKU Description	Notes	Segment	Gender	Manufacturer
1	Loreal		S:\CPG\Projects\E7186_Gille	SYSTEM RAZORS	FEMALE	
2	Head & Shoulders [1]		S:\CPG\Projects\E7186_Gille	SYSTEM RAZORS	FEMALE	
3	Head & Shoulders [2]		S:\CPG\Projects\E7186_Gille	SYSTEM RAZORS	FEMALE	
4	Generisk		S:\CPG\Projects\E7186_Gille	CARTRIDGES	FEMALE	
5	Palmers		S:\CPG\Projects\E7186_Gille	CARTRIDGES	FEMALE	
6	Pantene		S:\CPG\Projects\E7186_Gille	CARTRIDGES	FEMALE	
7	Dove		S:\CPG\Projects\E7186_Gille	CARTRIDGES	FEMALE	
8	Dove [2]		S:\CPG\Projects\E7186_Gille	CARTRIDGES	FEMALE	
9	Dubbledusch		S:\CPG\Projects\E7186_Gille	SYSTEM RAZORS	FEMALE	
10	Loreal [2]		S:\CPG\Projects\E7186_Gille	SYSTEM RAZORS	FEMALE	P&G

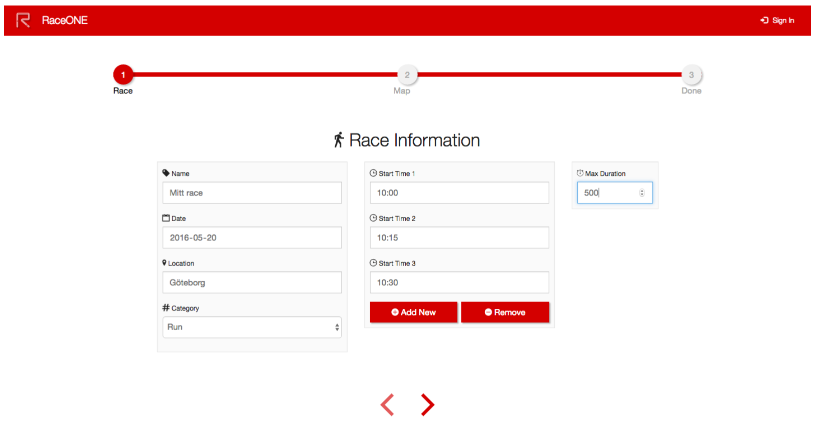
Figure A.1: Screenshot of the Skim module - SKU list view.



SKU List Hide matched

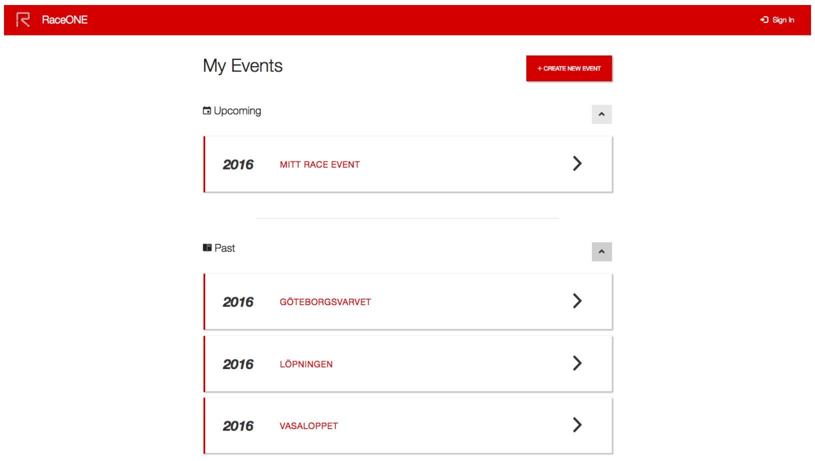
ID	Description	#Images
1	Loreal	22
2	Head & Shoulders [1]	23
3	Head & Shoulders [2]	14
4	Generisk	12
5	Palmers	9
6	Pantene	5
7	Dove	7
8	Dove [2]	3
9	Dubbledusch	3
10	Loreal [2]	3
11	Garnier	4
12	Garnier [2]	1
13	Garnier [3]	1
14	Garnier [3]	9
15	Garnier [3]	1

Figure A.2: Screenshot of the Skim module - Image SKU matching view.



Name: Mitt race
Date: 2016-05-20
Location: Göteborg
Category: Run
Start Time 1: 10:00
Start Time 2: 10:15
Start Time 3: 10:30
Max Duration: 500

Figure A.3: Screenshot of the Raceone portal - Create an event view.



Upcoming
2016 MITT RACE EVENT

Past
2016 GÖTEBORGSVARVET
2016 LÖPNINGEN
2016 VASALOPPET

Figure A.4: Screenshot of the Raceone portal - Event list view.

A.2 Additional Raceone Test Cases

Unfollow a participant
Unfollowing a participant removes the relationship created by following a user in a race.

Unparticipate from a race
This case will remove the relationship between a user and race, mainly an activity.

Fetch coordinates
This test will fetch all coordinates for a specific activity.

Remove coordinates
Removing coordinates will be done to optimize a route, or list of coordinates. This is done in batches, where IDs of the coordinates to be removed is sent to the database.

Fetch hot races
Fetching hot races consists of returning the 10 races with the most number of par-

ticipants and followers combined. This can be used by Raceone for sorting lists of races.

Remove race
A race in linked to several entities in the database - coordinates, event and activities where each activity has followers and coordinates. If this is done, it is usually when there are none or a few participants since it is made to rollback a mistake.