KYMENLAAKSON AMMATTIKORKEAKOULU

University of Applied Sciences

Double Degree Programme in Information Technology, specialization in Data Networks Engineering

Tomáš Panyko

UTILIZATION OF NOSQL

Bachelor’s Thesis 2015

ABSTRACT

KYMENLAAKSON AMMATTIKORKEAKOULU
University of Applied Sciences
Information Technology

PANYKO, TOMÁŠ	UTILIZATION OF NOSQL
Bachelor's Thesis	39 pages + 5 pages of appendices
Supervisor	Paula Posio, Senior Lecturer
Commissioned by	Kymenlaakson ammattikorkeakoulu, University of Applied Sciences
March 2015	
Keywords	Database, MongoDB, MySQL, Performance

This thesis is about explaining the use of NOSQL databases. It is a type of database focused primarily on storing a big data. It provides functions that are not available in standard relational databases, e.g. the possibility to store data in means other than the tabular relations.

This work was written as an extension of another bachelor thesis made at University of South Bohemia in Czech Republic to further explain the need of non-relational databases. The main objective was to show when, why and how to transfer relational database to one of the NoSQL database. At the beginning, the NoSQL databases are introduced. Further on, methods to transfer relational database to a non-relational version is explained. Next objective was to point out the advantages and disadvantages of the transferring. The last goal was to test the performance of the NoSQL database.

The content of this thesis is divided into two parts, theoretical and practical. In the theoretical part the main method to carry out the study was analysis of information resources. The practical part was about performance testing.

This thesis can be used as a study material and also can serve for future extension of Database study unit at Kymenlaakson ammattikorkeakoulu, University of Applied Sciences.

TERMS AND ACRONYMS

Big data	A wide term for so large datasets that traditional data processing is inappropriate.
Database schema	It is a structure, described by formal language, referring to the organization of data.
Dataset	It is a collection of data, usually corresponding to the content of a single table.
Index	It is a structure, designed to improve the speed of read operations.
NoSQL	Databases providing a different mechanism for storing and retrieving data than tabular relations in relational databases.
Relational database	Database based on the relational model, where data is in one or more tables
SQL	Structured Query Language is a programming language for managing data stored in a relational database.

TABLE OF CONTENTS

1 INTRODUCTION

The need of storing data is reaching far into the past. Humankind have always wanted to keep records, whether it is books, products from some e-shop or security numbers. Everything is stored in some way to be used in future. However, in some cases we are facing the difficulty of storing large amount of data. Imagine the scenario, where it is needed to keep millions of records of products and users. Every product has different properties, so it would be difficult to fit it into relational database with just tables and attributes. The important thing to realize is that, it is difficult, but not impossible. The truth is, it is possible to store almost everything in relational databases, but slow searching can occur and the feeling that compromises are made just to be able to store data. This can, of course, be a sign of poorly designed model, but it can also indicate that it is time to start looking for a different way of storing data.

This way is called NoSQL. The term covers especially column-oriented databases, key/value pair databases and document databases. It means that NoSQL is not just a database system, such as MySQL, but it is more like technology, a mechanism for storage and retrieval of data that covers wide variety of database systems other than tabular relations. The term NoSQL is often explained as "Not Only SQL" that suggests some of the database systems may support SQL query language, although it is rather rare come across such a system.

This thesis was written because of my long-lasting interest in databases. I have always admired the speed and elegance of searching and since my previous bachelor thesis called "NoSQL databases", I am more focused on alternative ways to store large amount of data, used by companies, such as eBay, Facebook and Amazon.

The main goal of this thesis was to show the reader that there are other ways of storing data. It explains when it is a good time to start thinking about using a NoSQL database and also helps to decide, which one of them suits his needs. Then it shows, how to properly transfer MySQL database to one of the most widespread NoSQL database, MongoDB. This transferring has some advantages and disadvantages, which were also covered in this topic. Test performance of MongoDB, showing the differences between MongoDB and MySQL, is covered in a later stage. This work expects that the reader has some knowledge of relational databases, although it is not necessary to understand the key aspects of this thesis.

2 NOSQL DATABASES

This chapter explains the basic idea of NoSQL including some key features of this technology. After that, the focus moves to the most widespread NoSQL databases.

2.1 Description

NoSQL databases are very different from SQL databases. While SQL is trying to keep an order, NoSQL is bringing a chaos. Users are often used to store their data in tables, where every table has attributes that each record must contain. This can be considered an advantage, but it also has some drawbacks. While it is helping to keep the records in a united form, it is limiting. Every record must have the same attributes to fit them to our table. There are strict rules that must be followed to secure correct behavior with almost no freedom in this model.

NoSQL databases, on the other hand, are going a different path. They are more open to unstructured data, e.g. data without defined data model. That means, you can store almost any information in your record. Since NoSQL is missing a join ability in queries in most cases, they are preferred in environments without relations among records, although it is possible to substitute this technique. This suggest most of the NoSQL databases are used for storing data without relations among them, such as log files or messages.

NoSQL databases support horizontal scaling, e.g. it is possible to spread your database across hundreds of servers to achieve better performance and high availability. Transactions in NoSQL do not offer full support of ACID (Atomicity, Consistency, Isolation, Durability) normally used in SQL, because of better availability and scaling. Most of the NoSQL databases do not support SQL query language. They are using a low-level query languages, such as C, Java or Erlang.

2.2 Main representative

As already mentioned, NoSQL databases are a mechanism for storing and retrieval of data, thus there are many options from which a user can choose. The most widespread are Column, Document, Key-Value and Graph databases.

2.2.1 Column databases

Instead of storing data in rows, as it is usual in SQL databases, column databases are storing data tables as series of columns of data, rather than as rows of data. Column stores offer very high performance and a highly scalable architecture. (1.)

To understand this model better, let us have a small example. In SQL, there are 2 dimensional tables, containing our data. Table 1 shows an example of this table, storing products.

id	category_id	description	price
1	3	Desc1	150
2	7	Desc2	200
3	2	Desc3	250

Table 1 Example of SQL table

Row oriented databases then join all the values in a row together. It is the same data, just simplified.

```
1, 3, Desc1, 150
2, 7 Desc2, 200
3, 2, Desc3, 250
```

Column oriented databases are working with columns instead of rows. The interpretation of this example would be:

```
1, 2, 3
3, 7, 2
Desc1, Desc2, Desc3
150, 200, 250
```

The structure of data seems to be just a simple opposite of row oriented databases. This approach has some benefits, though. When, for example, query SQL version for all `category_id` is necessary for statistics of products, it would require going through all rows to obtain a `category_id`. In column-oriented version, instead, all ids needed are already in one row.

HBase

HBase is an example of column oriented databases. It is developed as a part of Apache Hadoop project. It is an open source framework written in Java and designed to contain petabytes or even exabytes of distributed data. To store data across cluster, it is possible to use Apache Hadoop, which is an open source framework, containing Hadoop Distributed File System. HBase has been used for Facebook's messaging in 2010 due to incapability of MySQL to handle growing data sets. (2.)

2.2.2 Document databases

Document oriented databases are designed for storing, managing and retrieving document oriented data. The term "Document" does not have anything to do with a Word document or something similar, instead it is an encapsulated set of data, often represented as a key-value pairs. These pairs are encoded and called "Document". Encoding is often done by JSON or XML. An example of JSON document could be:

```
{
FirstName:”Tomas”,
LastName:”Panyko”
}
```

Which represents a document containing information that logically belongs together. In some cases, especially in MongoDB, one document can be thought as a one record from a table. Each document includes a unique key that represents that document.

MongoDB

MongoDB is a document oriented database that stores JSON documents in collections. Each database has a set of collections and each collection may contain many documents. MongoDB was developed in 2007 by 10gen Company. It is an open source project that became the most popular document oriented database and was adopted by many companies, for example eBay, Foursquare and the New York Times.

2.2.3 Key-Value databases

Key-value stores are considered the simplest NoSQL databases. Each record consists of a unique key and a value. Data is stored in an associative array, which is used as its data model. In this collection of key-value pairs, each key may appear at most once. This idea is often used in different types of databases, where richer data models are implemented on top of key-value stores.

Redis

Redis is the most popular key-value database. It is an open source, BSD licensed key-value cache and store. It supports variety of data types, such as strings, hashes, lists, sets and bitmaps. Companies that are using Redis include Twitter, Github, StackOverflow.

2.2.4 Object databases

In object databases, information is stored in form of objects, as it is in object-oriented programming. These databases combine both database capabilities and object oriented programming capabilities. They allow programmers to store their objects as they are in database. Query language is offered in most of the object databases, although no practical language is available. Object Query Language (OQL) was developed to fix this problem, but due complexity of object databases, no vendor fully implemented this language. There is no need of joins in accessing data, because objects are stored as a whole. Therefore object databases offer best performance when working with a lot of information from one object.

Db4o

Db4o is an open source object database written in Java and .NET. It can run on any operating system that has support for Java or .NET. Db4o is designed as embedded database, so it can be part of other software components. It supports Native Queries instead of OQL, allowing developers to use programming language, e.g. Java, C#, to access the database.

3 MONGODB

MongoDB was chosen as a representative for NoSQL databases and this thesis is mainly focusing on this solution. In this part, MongoDB is first described to understand the basic functioning with instructions, how to install and run this system. CRUD operations are necessary to operate the database and are part of this chapter as well.

3.1 Description

MongoDB is a document database, where every document belongs to one collection. A collection can be thought as a table in relational databases without the need of specifying, what information a table can contain. A document can be put into any collection, but it is advisable that every document in the same collection is similar.

BSON format is used for storing information. It is a binary-encoded serialization of JSON-like documents. BSON supports embedding of arrays and documents within other arrays and documents. BSON extends usability of JSON and supports also regular expressions, binary data and a date. Each document contains a unique identifier that is generated by MongoDB itself and serves for distinguishing documents.

MongoDB provides high performance data persistence. It supports embedded data models that reduces input/output activity on database. Embedding allows us to store relevant data in one place without joins with support of indexes. To provide high availability, MongoDB implements replica sets that is a group of MongoDB servers, maintaining the same data to provide redundancy and higher availability.

3.2 Installation and set up

MongoDB is available for wide variety of operating systems. From this link `http://www.mongodb.org/downloads` it is possible to download release for Windows, Linux, Mac OS X and Solaris. In this case, the focus will be on Windows version.

Installation is quite straightforward and requires only installation directory. These instructions assume installation to default directory `C:\mongod`.

MongoDB requires a directory for storing data. In default, the directory path is `/data/db`, although this directory is not present after installation. To proceed, it is necessary to create it. The directory path for storing data should be `C:\mongod\data\db.` It is also possible to change this default path to an alternative one. For this, there is an option `–dbpath` to `mongod.exe`. The result should look like this:

```
C:\mongodb\bin\mongod.exe --dbpath d:\database\data
```

MongoDB server is represented by `mongod.exe.` To start it, simply run `mongod.exe` from a command prompt, like this: `C:\mongodb\bin\mongod.exe`

If a directory that is used for storing data exists and no error occurs, a message `Waiting for connections` should be displayed at the bottom of the console output. This indicates that server is running correctly. Otherwise, an error message will be displayed in the console output, reporting what type of error occurred.

To connect to the database, open a new command prompt (`mongod.exe` must be still running) and type: `C:\mongodb\bin\mongo.exe`. This is a shell that will allow to work with a database system.

3.3 CRUD operations

CRUD operations is a shortcut for Create, Read, Update and Delete. They are basic tasks, needed to operate the database. In this section, they are going to be described with the commands to execute them.

3.3.1 Read operations

Read operations retrieve information stored in database. In MongoDB, a query retrieves documents from one collection. A query can specify certain conditions, which limits documents returned to the client. A query may also include a projection that specifies, which fields of documents should be returned. This leads to decreasing the amount of data that returns over the network in case of having a server and a client running on different machines.

For querying, MongoDB provides a method called `find()` triggered upon a collection. This method can stay empty without any parameters. In this case the result will contain all documents within chosen collection. It also accepts criteria and projections as parameters and returns a cursor to the corresponding documents. A cursor is an object that contains result of the query and can be iterated. It is also possible to include special methods, such as limits, skips and sort orders.

A query that returns _id, name and email of people that are older than 20 years old and limits this result to first five outcome could look like this:

```
db.clients.find( { age: { $gt: 20 } }, { name: 1, email:
1 } ).limit(5)
```

`Find()` method consists of two parameters. The first one specifies conditions that the result must meet, and the second parameter declares what fields should be returned. The query is constructed as a JSON document with keys and values.

To search for people older than 20 years, a query selection operator `$gt` is used, which stands for “greater than”. There are many operators to construct a query, for example `$lt` (“less than”), `$and`, `$or`, `$regex`. The whole list of operators can be found at http://docs.mongodb.org/manual/reference/operator/query/#query-selectors.

To specify what fields of results that meet the criteria should be returned, a projection can be used. It contains a list of fields to return or list of fields that should be excluded from the result. By saying `name: 1,` this field is included into the result, `name: 0` would exclude it.

There is one exception and that is `_id`. In the example above, the `_id` field is not mentioned among the projection and still it would be in the result. It is because `_id` is set to be returned by default and to exclude this field from result, `_id: 0` must be part of the projection.

3.3.2 Write operations

A write operation is any kind of operation that changes the data. In MongoDB, a write operation is always targeted upon single collection. All these writing operations are atomic on a document level that means if some error occurs during document modification, there cannot be any half modified documents. The document is either successfully modified or not modified at all.

There are three basic writing operations: insert, update and remove. Insert operation adds new documents to a collection. Update operation changes already existing documents and remove operation can delete documents from a collection.

3.3.2.1 Insert operation

Insert operation serves for adding new documents to a collection. To do so, there is a method `insert()` that can be called upon certain collection. An example of such method could be:

```
db.clients.insert(
{
         name: "Tomas",
         age: 26,
         email: "Panykot@gmail.com"
}
)
```

This method takes a document as its parameter and insert it into client's collection. The newly created document will have four fields: `name`, `age`, `email` and also `_id`, because every document must have `_id` and if it is not defined inside `insert` method, `_id` is generated automatically. In this case, `_id` is a unique `ObjectId` type that is 12-byte BSON based on timestamp, machine ID, process ID and an incremental counter.

3.3.2.2 Update operation

For updating documents in MongoDB, there is a method `update()` that can be called upon a collection. This method accepts query criteria to define what documents should be modified as well as options to further specify updating process, such as allowing multiple documents to be updated. An example of an update operation can look like this:

```
db.clients.update(
         { name: "Tomas" },
         { $set: { age: 27 } },
         { multi: true }
)
```

This method will search for every documents that include field `name` with value `"Tomas"` and change its `age` value to 27. If `age` field does not exist, it is created. By default, update operation can affect only one document, but with an option `multi`, it allows to change all documents that match the criteria.

If no document match the criteria, the collection stays without any change, unless `upsert: true` option is defined. In this case, an `insert` operation is performed, adding a new document into collection.

3.3.2.3 Remove operation

In MongoDB, there is a method `remove()` for deleting documents that can be called upon a collection. This method accepts a query criteria to specify, what documents should be removed. An example of this method could look like this:

```
db.clients.remove(
         { email: "Panykot@gmail.com" }
)
```

This command will search for every document that contain an `email` field with value `Panykot@gmail.com` and delete the whole document. In case of deleting only one document that match the criteria, the value 1 must be specified as a next parameter of `remove` function.

4 WHEN TO SWITCH YOUR DATABASE FOR A NOSQL

The important thing to understand is that it is generally not possible to say what database is better without knowing what data are stored and what is expected from the database. Decision to start using a NoSQL database instead of SQL should not be made rashly, because it requires a deep knowledge of our needs and also knowledge of what are the strengths and weaknesses of both concepts.

The nature of data is one of the primary indicators towards which concept works better for data storage and retrieval. Relational data are often referred as structured data, because tables and all its columns are defined before use. This approach brings reliability and stability, but also limitation to our database. On the other hand, NoSQL databases are schema-less, i.e. this approach offers more freedom in case the data are not structured, and therefore it is easier to handle changes in database.

The next indicator towards which concept should be chosen is the amount of data the database should store. It is no secret that NoSQL are aimed for larger datasets and by larger it means hundreds of millions of records, which are often semi-structured, generated from machines. This is managed by different way of scalability. While SQL is scaling vertically – adding more performance to a single node, NoSQL is scaling horizontally – adding more nodes to a system. SQL can scale very high, but it will eventually reach an upper limit. NoSQL can scale by adding more servers to a cluster and therefore scale almost infinitely.

Higher scalability and performance of NoSQL have a drawback though. By improving performance, some of the functionality is lost. It is not possible to develop a very fast NoSQL that has all the rich functionality SQL offers. NoSQL can be a smart choice, if higher performance is needed and it is not necessary to have all the SQL features, such as joins and transactions. Figure 1 shows, how functionality depends on performance and where MongoDB is situated.

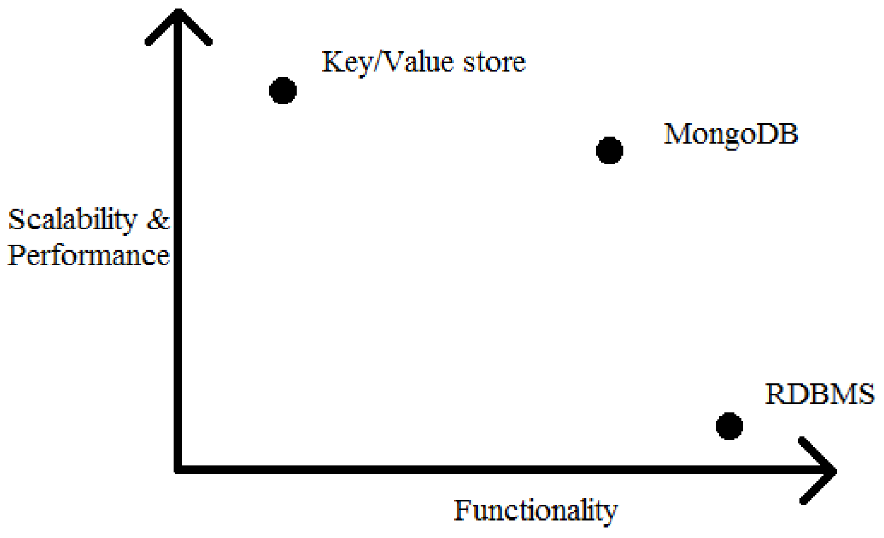


Figure 1 Performance vs. functionality

NoSQL databases are not meant for everyone and for every situation. They are focused on larger projects where data are not structured and performance is the priority. In many cases SQL is just enough to serve our needs. But as soon as our data start to grow and become more difficult to fit to predefined SQL tables, the time of looking for a NoSQL solution is in place.

5 CONVERTING MYSQL BLOG TO MONGODB

This section aims to demonstrate how a blog that stores its data in a MySQL database is converted to a blog that uses MongoDB. On a blog website there can be found some posts from redactors or admins and comments to those posts. Each post often belongs to some category and has some tags that describe its content. There can be some additional information such as the date of publishing and number of people who found the post interesting.

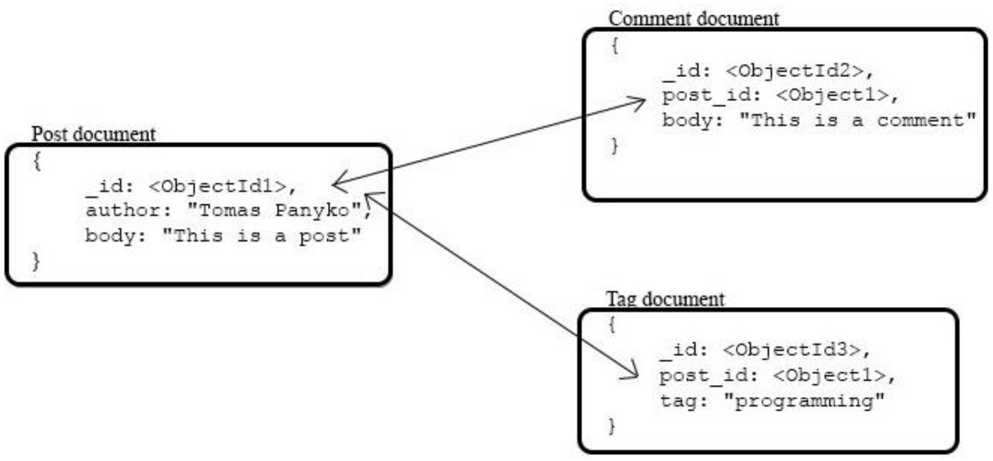
It is a very simple solution of blog that does not deal with authentication of redactors and users, instead it is focusing on performance. The database stores ten million posts and 100 million comments randomly divided among posts.

5.1 Choosing the right schema of data

The challenge in data modeling is about finding the needs of our application, performance of our database engine and the data retrieval patterns. While designing data model, it is very important to consider, how the application uses data for execution (queries, updates...) of the command and also take into account the structure of the data.

A very important part in designing data model for MongoDB is considering how to represent relations between data. In SQL world, there are tables with primary and foreign keys, which can be used to join particular tables and get the result. In MongoDB there are no tables nor joins. Instead there are references and embedding to express relations.

References are storing relationships between documents by adding links from one document to another. This approach is very similar to primary/foreign keys in SQL world. As mentioned above, MongoDB does not have joins to connect documents, therefore in case of making relationships by references, joins must be done in an application layer. Relationship by references are considered as normalized data models. Figure 2 shows references among three collections – Post, Comment and Tag.



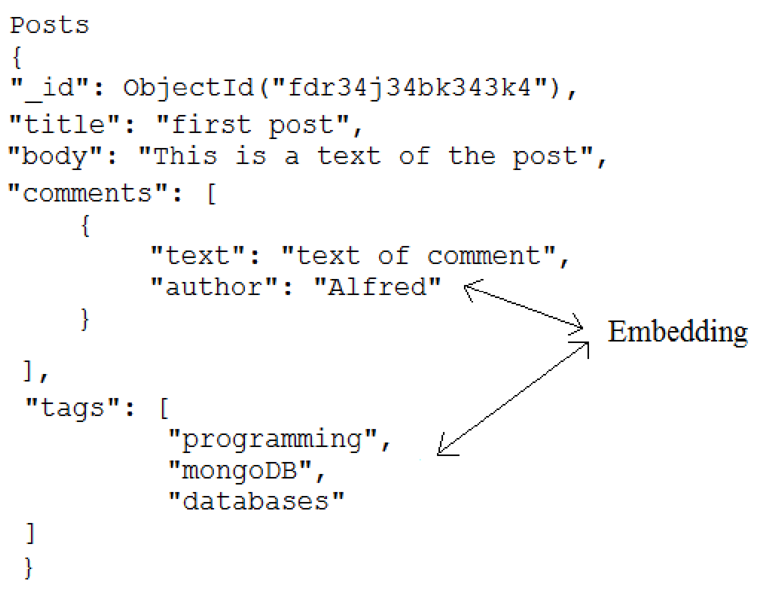
Post document
```
{
    _id: <ObjectId1>,
    author: "Tomas Panyko",
    body: "This is a post"
}
```

Comment document
```
{
    _id: <ObjectId2>,
    post_id: <Object1>,
    body: "This is a comment"
}
```

Tag document
```
{
    _id: <ObjectId3>,
    post_id: <Object1>,
    tag: "programming"
}
```

Figure 2 Normalized data model (3)

On the other hand, embedding or embedded documents are dealing with relationships between data by storing related data within one document. MongoDB allows to store subdocuments or arrays within a document. Embedding is referred as a denormalized data model and with this approach an application can manipulate with related data in one atomic database operation without the need of accessing other collections.



```
Posts
{
"_id": ObjectId("fdr34j34bk343k4"),
"title": "first post",
"body": "This is a text of the post",
"comments": [
    {
        "text": "text of comment",
        "author": "Alfred"
    }
 ],
 "tags": [
          "programming",
          "mongoDB",
          "databases"
 ]
 }
```

Figure 3 Denormalized data model (3)

In general, using normalized data models is good when embedding would result in large amount of duplicated data, but would not provide higher performance to outweigh the consequences of duplication. A denormalized data model, on the other hand, is more suitable for situations, where storing one-to-one or on-to-many relationships. In general, a denormalized data model provides better performance for read operations and the ability to retrieve related information with a single query.

Since the blog that will be transferred from MySQL to MongoDB database will not result in a large duplication after embedding, denormalized data model seems to be the better choice, although it has many-to-many relationships. Moreover, in case of having a normalized data model, it would have to access all the collections and join the documents, making the query slow and complicated.

5.1.1 SQL schema

It is a good practice to have an SQL database in 3. NF, if possible. With this approach duplication is reduced and referential integrity ensured. As shown in Figure 4, having database in 3. NF results in six tables. Since each post can have multiple tags and categories and each tag and category can be used more than once, it would result into many-to-many relationships that are difficult to implement in SQL without breaking tables into smaller parts.

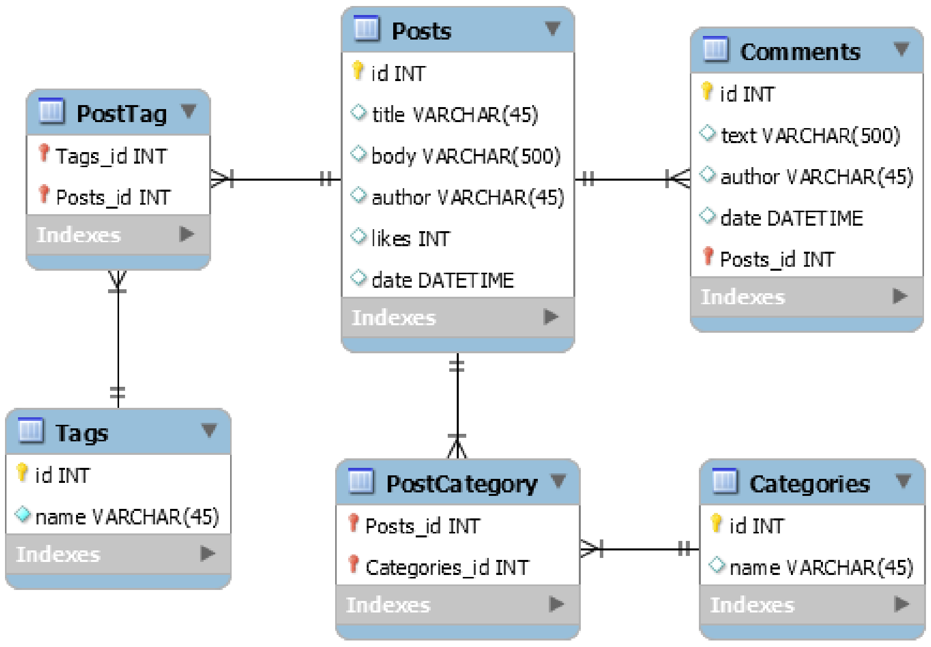


Figure 5 SQL diagram

In this example, there are four main tables and two additional. Main tables are Posts, Comments, Tags and Categories. Table Posts is storing basic information about every post, such as title of the post that can be used as a headline, body that contains the text of the post itself, author that is just represented as a string without additional table, likes that represent, how many people clicked some button on a webpage and a date, saying when the post was published.

Table Comments stores information about comments, such as text of the comment itself, author of the comment, again without any special registration table for that, therefore an user will just simply fill his name into a textbox. Then the table Comments stores information about date, when a comment was published and most importantly an id of the post that the comment belongs to. Tables Tag and Category are simple and contain only name of tag and category. Tables PostTag and PostCategory serve as bridges between Post-Tag and Post-Category tables. Each table contain foreign keys of two tables (i.e. table PostTag has foreign keys of Post and Tag table) to allow having multiple categories/tags in one post.

5.1.2 MongoDB schema

After realizing what is better for this example of blog, references, embedding or using both of them, it is very simple to design this schema. With embedding, there are duplicates of tags and categories, but also higher performance.

This schema is using only one collection, which is called Posts. Each document within Posts contain complete information about single post. That include title, which represents a headline of the post, date of post publication, author of the post, how many likes have the post got, text of the post, an array of comments, where every comment has a text, an author and a date. At the end there are two arrays, one of them is for storing tags and the second one is for storing categories. With this solution it is possible to store all information in one place with no need of connecting different collections. An example of this document is shown on Figure 5.

```
Posts
{
"_id": ObjectId("fdr34j34bk343k4"),
"title": "first post",
"date": "ISODate("2015-02-26..."),
"author": "Tomas Panyko",
"likes": 0,
"body": "This is a text of the post",
"comments": [
    {
        "text": "text of comment",
        "author": "Alfred",
        "date": "ISODate("2015-02-26...")
    },
    {
        "text": "second comment",
        "author": "Peter",
        "date": "ISODate("2015-02-26...")
    }
],
"tags": [
         "programming",
         "mongoDB",
         "databases"
],
"category": [
         "Development"
]
}
```

Figure 4 MongoDB schema

5.2 Advantages and disadvantages of converting to MongoDB

Using MongoDB instead of relational database brings some advantages and disadvantages, because they are completely different systems that require different way of thinking and are meant for different situations. It is up to the DBA to consider if pros really outweigh cons in his project.

5.2.1 Advantages of using MongoDB

1. Sharding and Load balancing

Sharding is a way of storing data across multiple machines in MongoDB. As the amount of data is increasing, approach of storing the data on a single machine may not be sufficient and will not offer needed read and write throughput. Sharding solves this issue with horizontal scaling. Sharding allows to add more computers to support data growth and requirements of read and write operations. (4.)

When working with large amount of data, they are usually distributed across multiple machines. To distribute data more equally, so none of the machines is overloaded, MongoDB implements a load-balancer. This feature is enabled by default and is more advanced than load balancing in MySQL. (5.)

2. Speed

MongoDB querying is usually much faster than in MySQL, especially when data are stored in one collection using embedding and can be retrieved in a single query. However, in case of using many references, which essentially simulates a relation model with primary/foreign keys, speed of queries in order to retrieve a document will drop and sometimes can be even slower than a relational solution.

3. Flexibility

MongoDB is a schema-less database, therefore does not require unified structure of data across collection. This is an advantage over relational model, because implementing of changes is done easily just by modifying data. However, consistency of data is an important matter. So, whenever a collection is created, data with identical or at least very similar structure should be stored there.

5.2.2 Disadvantages of using MongoDB

1. No Joins

MongoDB does not implement joins as known from relational database. To simulate this, references must be implemented, multiple queries made, and then the data joined within an application. However, this can often lead to slow and complicated codes that can make using of MongoDB very painful.

2. Duplication

Since there are no joins, there is a chance to embed documents that would have to be otherwise separated among multiple collections. However, this approach often brings duplication of data. It is up to the administrator to decide if the amount of duplication is tolerable or if better model is needed.

3. Lack of multi-document transactions

Operations that are performed on a single document are always atomic in MongoDB. Operations that involve multiple documents, often called as transactions, are not atomic, though. Since MongoDB supports embedding, documents can include multiple subdocuments within and can become fairly complex, single document atomicity offers sufficient support for many use cases. (6.)

6 SHARDING AND REPLICATION

One of the difference between SQL and NoSQL solution is the way they are scaling their data. While SQL is using vertical scaling, NoSQL sticks with horizontal scaling. In world of MongoDB, this is made by sharding. To provide higher availability of data, replication is often implemented together with sharding. This chapter tries to explain these techniques. Since they are meant mainly for production environment and not for testing, they are not implemented in the practical part of this thesis.

6.1 Sharding

Sharding is a method that allows to store data across many machines. MongoDB supports sharding of very large data sets and high throughput operations. These conditions can be a challenge for a single server. High number of queries can fully stress the CPU of a server and large data sets may be bigger than storage capacity. An example of shading is shown on Figure 6.

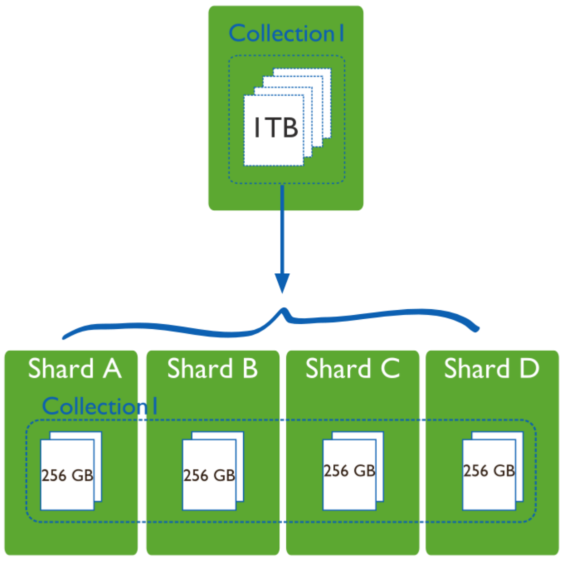


Figure 5 Sharded collection (7).

With sharding, it is possible to divide data sets and store the data in multiple servers. Every shard is an independent database and together they make the whole logical database.

To ensure higher throughput, sharding lowers the number of operations that each shard must take care of. If a record for updating is located on Shard A, the application will access only that specific shard and not the others.

Sharding also lowers the storage requirements of every shard. If a cluster contains four shards and the whole data set has two terabytes, then each shard might hold only 512GB. It is much less than with just two shards, where each would have to store one terabyte. In sharded cluster there are three main components: shards, query routers and config servers.

Shard serves for storing data. In case higher availability and data consistency is needed, each shard can be a replica set. More about this in a following section. Query router, also called a mongos, stands between client application and shards. It is an interface for communication with shards that can be used for direct operations. The query router processes operations on shards and returns the result to the user. Config server stores metadata, which is mapping cluster's data set to all shards. The query router is using this config server to target operations to appropriate shards. An example of sharded cluster with all the main components is shown on Figure 7.

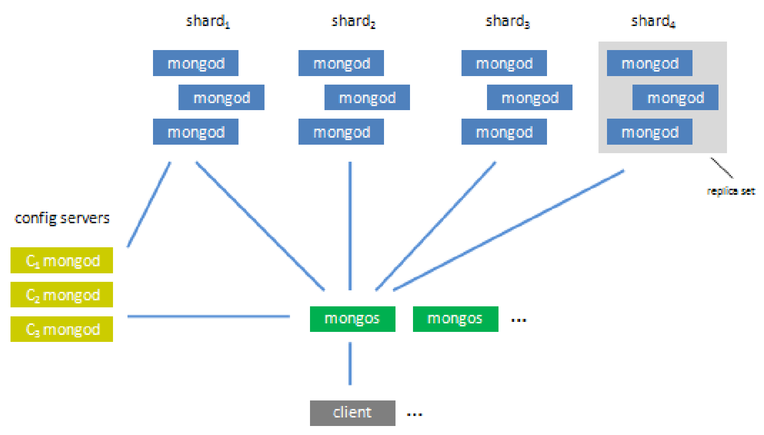


Figure 6 Sharded cluster (8)

6.2 Replication

For redundancy and higher data availability there is a technique called replication. Replication protects database against the loss of a server by storing copies of data on different machines. It can be also used for increasing read performance. User can then read and write on different servers. Maintaining multiple copies of data in different geographical locations, for instance datacenters in different countries, may also improve availability of data for distributed applications.

In MongoDB, a replica set is a group of mongod with the same data. One mongod, called primary, receives all write operations from the user. Rest of the instances then read an oplog file, which contains all operations that led to changes on primary mongod and apply those operations as well, so that the whole replica set has the same data.

In replica set there can be only one primary mongod. This mongod takes care of all write operations from clients. By default, primary mongod is managing all the reads from database, therefore it provides strict consistency of data. Figure 8 shows, how a client application communicates with MongoDB in case of replication.

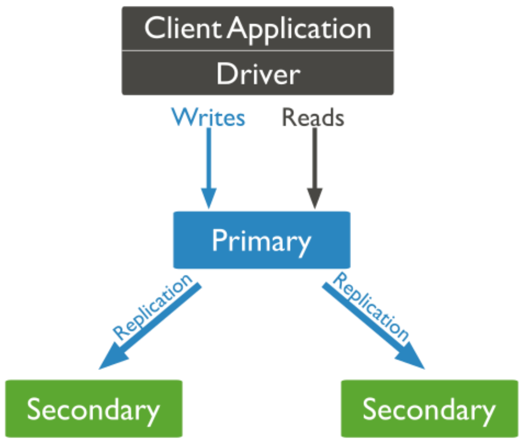


Figure 7 Primary mongod in replica set (9)

Other mongods within replica set are called secondary. These secondaries copy primary’s oplog and perform operations within this file. This way they can contain the same data as primary mongod has. If primary mongod becomes unavailable, replica set will elect a new primary from secondaries. By default, reads are directed on primary mongod, however, a user can change read preferences and allow reading from secondaries. This can lead to returning of data that are not up-to-date, because synchronization between primary and secondaries can take some time.

7 PERFORMANCE TESTING

This section is focusing on testing both MongoDB and MySQL with Innodb storage engine databases. Testing consists of basic CRUD operations (create, read, update, delete). In every test the main focus is on operations that would be probably needed the most, to better simulate needs of application. However, before this is going to happen, the database needs some data to operate with. The subchapter “Creating dataset“ is focusing on how to create a file containing data and how to import them to the database.

7.1 Testing environment

Testing environment often determines, how fast the database can be. All the following testing was performed on a personal laptop, which is not the best solution, but still can provide interesting output.

Operating system	Windows 7 SP1 64bit
Processor	Intel Core i3-3110M 2.40GHz
Memory	4GB RAM
Hard disk	SATA SSHD 500GB
SQL Server	MySQL 5.6.17, using Innodb storage engine
MongoDB server	MongoDB 2.6 Standard

Table 2 Testing environment

7.2 Creating dataset

Each database require different shape of data. MongoDB is working with JSON and supports importing JSON documents from files. MySQL, on the other hand, can import data from .csv files.

To create both files containing data, Java language has been used to generate random data and store them in an appropriate shape. These files are then used as an input of database. For importing data to MongoDB, there is a special application called `mongoimport` that is a part of installation package. MySQL is using `LOAD DATA INFILE` statement to read rows from a text file into a table at a very high speed. Both databases are storing ten millions of posts and 100 millions of comments.

7.2.1 Dataset for MongoDB

MongoDB requires JSON documents in order to import them to the database. For better cooperation of Java and JSON, there is a library called ”json-simple”, which is accessible on https://json-simple.googlecode.com/files/json-simple-1.1.1.jar. The main code for creating dataset consists of for cycle that is executed ten million times. In each cycle, a unique document in generated with random values. Each document has its own _id, title, date, author, likes, body and array of comments, tags and categories. This document is then saved to a JSON file and the cycle then starts over again. The whole class needed for creating dataset is in Appendix 1.

After the dataset file is created, `mongoimport` can be used to import data into database. To do so, database server must be running. Importing can be easily done by the command:

```
mongoimport --collection posts --file posts.json
```

This imports the content of the file `posts.json` into the collection named `posts.` After the import is done, the data is stored within the database.

7.2.2 Dataset for MySQL

MySQL dataset, as a dataset for MongoDB, is also created in Java. Since MySQL is using tables, each created file represents data of one table. There are totally six tables in Blog project, therefore six dataset files are needed. Because some files would be very large, table comments contains one hundred millions of records, it is a good practice to divide a dataset to smaller parts. In this case, dataset for comments is divided into ten files, each containing ten millions of records. With this approach a MySQL is able to achieve higher insert performance. Each file is stored as a CSV file, where values are separated by semicolon and every row represents one record. The class needed for creating all the datasets is in Appendix 2.

MySQL has a `LOAD DATA INFILE` statement that can be used for importing data into database. To be able to do that, MySQL server must be running and client must be connected to the server to issue this command. An example of this command for loading posts into database could look like this:

```
LOAD DATA LOCAL INFILE 'D:\Posts.csv' INTO TABLE posts
FIELDS TERMINATED BY ';' LINES TERMINATED BY '\n' (title,
body, author, likes, date);
```

First it is specified, where the file is stored and then, which table should contain data. After that a character for a field separation is defined together with a character determining a new record (\n means new line). At the end there is a list of table columns that will be used for the dataset.

7.3 Insert test

The insert test is focused on the time needed for importing data into the database. Both databases have certain ways for importing large amount of data. In MongoDB there is an application called `mongoimport`, which is located in bin folder within the installation path. MySQL has a statement `LOAD DATA LOCAL INFILE`, which can be called from MySQL client. Both databases are importing ten millions of posts and 100 millions of comments. Each post can also have multiple tags and be in multiple categories.

7.3.1 Insert test in MongoDB

MongoDB was importing a JSON document, which contains ten millions of posts with embedded comments. Settings of MongoDB server was not modified in any way, since the default settings is very well tuned.

To start importing data into the database, `mongod` instance must be running. If it is, open a command prompt and locate the installation folder. Within bin folder there is the `mongoimport.exe` application. Start the application by this command:

```
mongoimport --collection posts --file posts.json
```

It launches immediately importing. All posts are then stored in the posts collection. Since all the relative information to each post are stored in one document, it also improves insertion time. This process could be also executed by `insert()` method within java code, but `mongoimport` offers better performance for importing larger amount of data.

MongoDB needed 58 minutes and 12 seconds to complete this task. The speed was mostly constant during the import, showing no problems with larger datasets.

7.3.2 Insert test in MySQL

MySQL was working with multiple CSV documents, each of them representing certain table in database. In case of comments table, the dataset was divided into ten smaller CSV files to improve insertion speed. This test consisted of filling all the tables within blog database. Each table has the number of records as shown in Table 3:

Table	**Number of records**
Posts	10000000
Comments	100000000
Tags	10
Categories	5
PostTag	34875000
PostCategory	16380698

Table 3 Number of records in tables

In default settings MySQL provides reasonable performance that is enough for most cases. For importing larger amount of data, the settings in not optimal though. Therefore some of the settings were modified to achieve higher insert speed. The changes are listed below.

```
innodb_buffer_pool_size = 1G
autocommit = 0
unique_checks = 0
foreign_key_checks = 0
```

`Innodb_buffer_pool_size` sets the amount of memory where InnoDB caches table and index data. The larger the amount is, the less disk I/O is needed.

`Autocommit` tells the server to commit every change in the database as soon as it is executed. Value 0 turns this feature off and allows bulk loading.

`Unique_checks` with value 0 speeds up the import by turning off the uniqueness checks during the import operation.

`Foreign_key_checks` with value 0 disables checking of foreign key constraints.

For inserting data into the database a `LOAD DATA LOCAL INFILE` statement has been used for each table. An Insert operation can be used as well, but it is considerably slower.

The time needed to complete this task was 8 hours 23 minutes and 39 seconds. The slowest was filling the comment table, since the process was slowing down as the table was growing. The bad result can be explained by using a laptop with just 4GB of RAM and hard drive using magnetic memory. Moreover, additional changes in settings could lead to better performance.

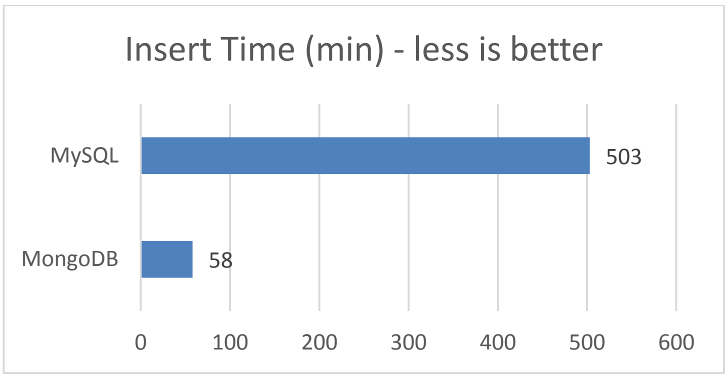


Figure 8 Insert time

7.4 Search test

The search test performs queries to determine, how fast both databases can get results. Each of them has a client that can communicate with server and query it. The focus is aimed on queries that could be often used and therefore simulate real needs.

7.4.1 Search test in MongoDB

First test queried server, asking for first 50 posts having the tag ”MongoDB”. This can be fairly frequent query as far as application provides clickable tag buttons. A user then can easily filter posts that are related to his search. The query in this case is very simple, since all the relevant data are in one collection.

```
db.posts.find({tags:”MongoDB”}, {_id:1}).limit(50)
```

First it is specified that the query is looking for a tag with value “MongoDB”. It does not matter if the value is in an array or not, the query goes through every field called “tags” and search for specified value. Second part of find method tells the server to return only the id of record that match. Limit function then restrict the result just for first 50 matches.

The time needed to perform this query was 0.032 sec. It is important to keep in mind that there is only one index in default and that is _id, which was not used in this search, because the query was looking for a value in the tags field. To further test the ability of MongoDB, tags field was indexed. The command needed to create a second index on tags field is:

```
db.posts.ensureIndex({tags:1})
```

This will create an index on the tags field in ascending order. After applying this command and querying the server again, the search time improved to 0.005 sec.

In second test the query was searching for first 50 posts, where “Veronique Mather” added some comments. This query can be used for tracking user comments and for users control panel. The command is again very simple:

```
db.posts.find({"comments.author": "Veronique Mather"},
{_id:1}).limit(50)
```

To perform this query, the server needed 15.509 seconds. This result is again without an index on the author field within comments. To improve it, the index was applied the same way as in previous test:

```
db.posts.ensureIndex({"comments.author":1})
```

With this change, the time needed to find first 50 posts was reduced to 0.109 sec. Applying index speeded up the query 149x.

7.4.2 Search test in MySQL

First test, as in MongoDB, was querying database for first 50 posts with tag “MongoDB”. This query joins 3 tables – posts, tags and posttag, looks for a tag name “MongoDB” and limits the output to first 50 matches.

```
SELECT posts.id FROM posts INNER JOIN posttag ON posts.id
= posttag.Posts_id INNER JOIN tags ON posttag.Tags_id =
tags.id WHERE tags.name="MongoDB" LIMIT 50;
```

Thanks to the fact that foreign keys are indexed, the search speed was fast, despite the size of the tables. The time needed to perform this query was 0.140 sec. Comparing to results of MongoDB, MySQL was approximately 4x slower and after applying index on tags field, MySQL was slower even 28x. This result suggests that embedding has better read performance over joining tables in MySQL.

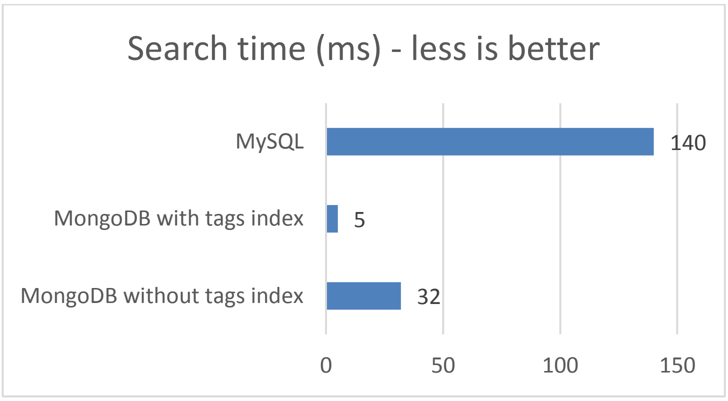


Figure 9 Search time

The second test targeted comments. The task was to find posts, which were commented by the user “Veronique Mather” and limits the output to first 50 results. This query joins two tables – posts and comments and then search in comments for the author “Veronique Mather” and returns first 50 hits.

```
SELECT posts.id from posts INNER JOIN comments ON com-
ments.Posts_id = posts.id WHERE comments.author = "Ve-
ronique Mather" LIMIT 50;
```

The query performed very poorly, taking 4 minutes 39 seconds to finish. This is mostly because of joining the two largest tables in the database – posts with ten millions of records and 2GB of data and comments with 100 million of records and 22GB of data. MongoDB was again better in this test, first attempt to query the database without the index on author was 18x faster and with the index 2559x faster, proving that MySQL had serious problems while performing queries on the tested machine.

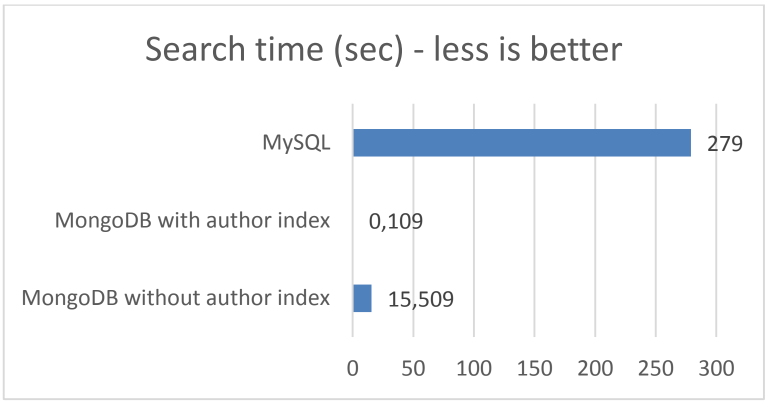


Figure 10 Search time

7.5 Update test

In this test, the focus is on update speed of both databases. The test consists of finding records that match the criteria and changing some of their values.

7.5.1 Update test in MongoDB

The task of this test was to determine, how fast MongoDB can change posts, where author is ”Veronique Mather”. If it finds such a post, it changes its body to ”modified text” and sets its date to current date and time. It simulates updates of posts from certain author and combines read and write capabilities. The update command is more complex in comparison to find method.

```
db.posts.update({author:"Veronique Mather"},{$current-
Date:{date:true}, $set:{body: "modified text"}},
{multi:true})
```

First part specifies, what to look for. It is equal to WHERE statement in MySQL. Second part specifies, what should be changed and to what value. That is equal to SET statement. Third part is options settings. In this case the command tells the server to update every match in the database.

In the whole collection there are ten million of posts and 22142 of them were written by Veronique Mather. To find all of them and change them, MongoDB needed 3 minutes and 5 seconds without an index on author field. After applying it, the time improved to 2 minutes and 17 seconds. The command to set the index on the author field is:

```
db.posts.ensureIndex({author:1})
```

7.5.2 Update test in MySQL

MySQL had the same goal as MongoDB – find every post that has been written by Veronique Mather and change its body to “Modified text” and date value to current date time. It works only with table posts, hence there are no joins. The command is very simple and clearer than MongoDB’s version.

```
UPDATE posts SET body="modified text", date=NOW() WHERE
author="Veronique Mather"
```

In this test, MySQL achieved very interesting result. Since there are no joins between tables, MySQL was able to work very fast and changed all the matching records in 2 minutes and 36 seconds. It proves that MySQL is able to work with similar performance as MongoDB, as long as it works with only one table.

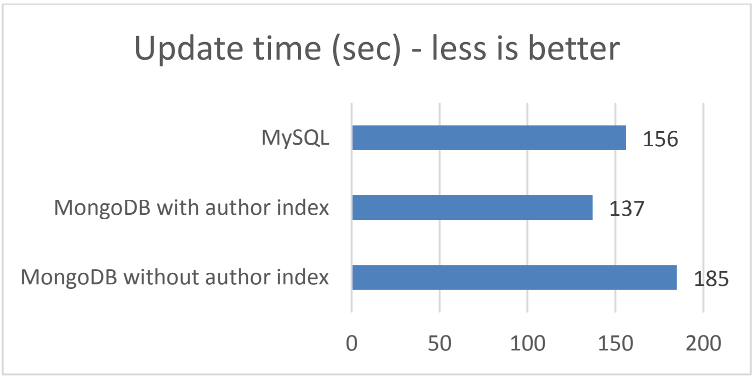


Figure 11 Update time

7.6 Delete test

Final part of performance testing is focusing on deleting data from both databases. Each database is tasked to delete certain data that match the criteria.

7.6.1 Delete test in MongoDB

Both databases got the same task – To delete all the posts written before 2012, including all the related comments. This can be used for database maintenance, when an administrator decides to remove old posts, because they are no more important and make then the database lighter and faster. The command to make it happen in MongoDB is very simple:

```
db.posts.remove({"date":{"$lt":ISODate("2012-01-
01T00:00:00.000Z")}})
```

Remove method accepts one parameter, telling database, what to look for. In this case it looks for the field date and if its value is less than the specified value of ISODate, the whole document, storing related information to the post, is removed.

In the collection of posts there are nearly 4 million of posts that match the criteria. The time needed to find them and remove them was 157 minutes. This time was achieved without an index on date field. After restoring data to the previous version and applying the new index, the time needed to perform this command changed to 182 minutes. These poor times are results of low amount of RAM, where data nor index cannot fit into memory and are stored on hard drive instead. Index then even slows the delete process due to the need of removing it. The command to add an index on date field is:

```
db.posts.ensureIndex({date:1})
```

7.6.2 Delete test in MySQL

MySQL had to delete all the posts that are older than 2012 as well. This situation is different comparing to previous tests, though. Since posts related information are spread across multiple tables, deleting must reach them all and remove all the records that are related to the matching post, namely tables posts, comments, posttag and postcategory. To do so, there is a trigger function.

```
CREATE TRIGGER deletePosts BEFORE DELETE on posts
FOR EACH ROW
BEGIN
DELETE FROM comments WHERE comments.Posts_id = old.id;
DELETE FROM posttag WHERE posttag.Posts_id = old.id;
DELETE  FROM  postcategory  WHERE  postcategory.Posts_id  =
old.id;
END
```

This function makes sure that before deleting a post, all the related records from comments, posttag and postcategory are removed. After applying this function, a command to start removing matching posts can be safely called.

```
DELETE FROM posts WHERE YEAR(date) < 2012
```

Delete command is very simple. It extracts year from date value and compare it to 2012. If the value is lower, the trigger is called. After it is finished, the record from posts is deleted.

This task proved to be over the capability of MySQL. After one day of running, the command had to be cancelled, because the estimate time to finish it was 3 days. InnoDB has performance issues when deleting large amount of data from large tables. The solution of that could be divide the data to smaller chunks, transferring the engine to MyISAM or modifying the settings. Both databases proved to have problems with deleting data. In MongoDB the problem was mostly in the available RAM, while MySQL had more problems with the engine and settings itself.

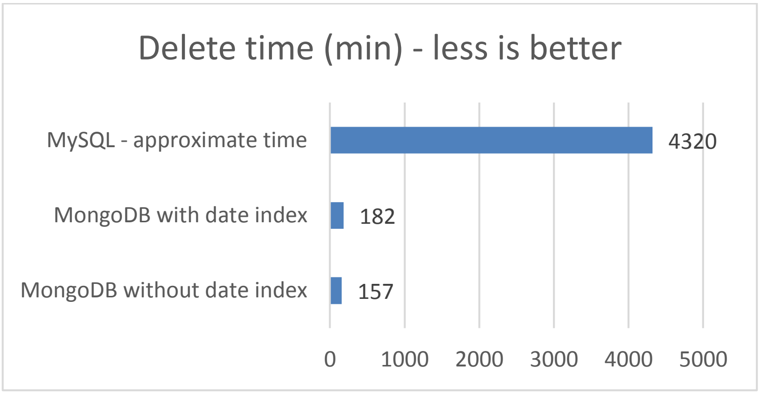


Figure 12 Delete time

8 CONCLUSION

The goal of this thesis was to introduce NoSQL databases and their utilization. In the theoretical part the focus was on explaining, what NoSQL means and introducing the main players on NoSQL field. Later on, one of them, MongoDB, was chosen to demonstrate differences between relational and non-relational databases. First it is explained, how MongoDB works, how to install it and what commands can be used to operate the database. Because NoSQL is not meant for every situation, it is described when a database administrator should consider transferring his relational database to non-relational version, how to actually do it and what will be the benefits and drawbacks of such action. Although sharding and replication are not implemented in the practical part, it is considered a very important feature of MongoDB and therefore worth mentioning it.

The practical part was about performance testing. Both MySQL and MongoDB databases were designed and in java language created datasets for each of the database. Databases were storing data from fictional blog site. Each database contained ten million posts and 100 million comments to those posts. Performance was tested on insertion of data, searching the database, updating records and deleting them. All the tasks were designed to simulate real situations, which can be common for a blog site.

At the end of this thesis I would like to share my opinion on NoSQL. It is important to understand that NoSQL is only suitable for certain situations and cannot never fully substitute SQL. Some people say that NoSQL is useless and does not bring any real advantages over SQL. From my point of view, I can understand them. In most cases, databases are storing limited amount of data and the structure does not change often. You will not probably need a NoSQL solution for that. It could be actually contra productive. However, then you come across a larger project, where data is changing, adding more attributes and making it more challenging to store it within tables. In these cases, NoSQL has its place. It is a solution for situations, when relational model is losing its breath.

SOURCES

1. NoSQL Databases Defined & Explained | Planet Cassandra. 2015. *NoSQL Databases Defined and Explained* [online]. Available from: http://planetcassandra.org/what-is-nosql/ [Accessed: 19 January 2015].
2. The Underlying Technology of Messages. 2010 [online]. Available from: https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919 [Accessed: 25 January 2015].
3. MongoDB Manual. 2015. *Data Model Design - MongoDB Manual 3.0.2* [online]. Available from: http://docs.mongodb.org/manual/core/data-model-design/#data-modeling-referencing [Accessed: 11 February 2015].
4. MongoDB Manual. 2015. *Sharding - MongoDB Manual 3.0.2* [online]. Available from: http://docs.mongodb.org/manual/sharding/ [Accessed: 20 February 2015].
5. Valhalla Articles - The Pros and Cons of MongoDB. 2014. *The Pros and Cons of MongoDB* [online]. Available from: http://halls-of-valhalla.org/beta/articles/the-pros-and-cons-of-mongodb,45/ [Accessed: 25 February 2015].
6. MongoDB Manual. 2015. *Perform Two Phase Commits - MongoDB Manual 3.0.2* [online]. Available from: http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/ [Accessed: 27 February 2015].
7. MongoDB Manual. 2015. *Sharding Introduction - MongoDB Manual 3.0.2* [online]. Available from: http://docs.mongodb.org/manual/core/sharding-introduction/ [Accessed: 02 March 2015].
8. Automating partitioning, sharding and failover with MongoDB - Server Density Blog. 2010. *Automating partitioning, sharding and failover with MongoDB* [online]. Available from: https://blog.serverdensity.com/automating-partitioning-sharding-and-failover-with-mongodb/ [Accessed: 10 March 2015].
9. MongoDB Manual. 2015. *Replication Introduction - MongoDB Manual 3.0.2* [online]. Available from: http://docs.mongodb.org/manual/core/replication-introduction/ [Accessed: 02 April 2015].

```
public class DataSetMongo {
    public static void main(String[] args) {
        createDataSet();
    }
    static String[] tags = {"Programming", "MongoDB", "Databases",
"Linux", "Windows", "Hacking", "VMWare", "VOIP", "CPU", "Harddisk"};
    static String[] categories = {"Development", "Gaming", "Secu-
rity", "Operating system", "Mobile devices"};
    static String text = "Lorem Ipsum is simply dummy text of the
printing and typesetting industry. Lorem Ipsum has been the indus-
try's standard dummy text ever since the 1500s, when an unknown
printer took a galley of type and scrambled it to make a type speci-
men book";

    private static void createDataSet()
    {   ArrayList names = new ArrayList();
        try(BufferedReader br = new BufferedReader(new File-
Reader("D:\\DataSetNames.txt"))) {
        String line = br.readLine();
        Random r = new Random();
        int numberOfPosts=0;
        JSONObject obj1;
        JSONArray list1 = new JSONArray();
        int count=1;
        while (line != null)
        {
            names.add(line);
            line = br.readLine();
        }
        for(int i=0; i<10000000; i++)
        {
          JSONObject obj = new JSONObject();
          obj.put("_id", i);
          obj.put("title", i+". post");
          JSONObject obj6 = new JSONObject();
          obj6.put("$date","201"+r.nextInt(5)+"-0"+(r.nex-
tInt(8)+1)+"-"+r.nextInt(2)+""+(r.nextInt(8)+1)+"T"+r.nextInt(1)+
r.nextInt(9)+":"+r.nextInt(5)+r.nextInt(9)+":"+r.nextInt(5)+r.nex-
tInt(9)+".483-0400");
          obj.put("date", obj6);
          obj.put("author", names.get(r.nextInt(names.size())));
          obj.put("likes", r.nextInt(300));
          obj.put("body", text.substring(r.nextInt(text.length())));

          if(count ==1)
          {
            numberOfPosts=r.nextInt(20);
             count++;
          }
          else
          {
            numberOfPosts=20-numberOfPosts;
            count=1;
          }
          for(int j=0; j<numberOfPosts;j++)
          {
             obj1 = new JSONObject();
             obj1.put("text", text.substring(r.nex-
tInt(text.length())));
             obj1.put("author", names.get(r.nextInt(names.size())));
             JSONObject obj4 = new JSONObject();
             obj4.put("$date","201"+r.nextInt(5)+"-0"+(r.nex-
tInt(8)+1)+"-"+r.nextInt(2)+""+(r.nextInt(8)+1)+"T"+r.nextInt(1)+
r.nextInt(9)+":"+r.nextInt(5)+r.nextInt(9)+":"+r.nextInt(5)+r.nex-
tInt(9)+".483-0400");
             obj1.put("date", obj4);

             list1.add(obj1);
          }
        JSONArray list2 = new JSONArray();
        int number= r.nextInt(tags.length);
        for(int j=0; j<number; j++)
        {
            String value = tags[r.nextInt(tags.length)];
            if(!list2.contains(value)) list2.add(value);
        }

        JSONArray list3 = new JSONArray();
        number= r.nextInt(categories.length);
        for(int j=0; j<number; j++)
        {
            String value = categories[r.nextInt(categories.length)];
            if(!list3.contains(value)) list3.add(value);
        }
        obj.put("comments", list1);
        obj.put("tags", list2);
        obj.put("categories",list3);
           try {
                      FileWriter file = new File-
Writer("d:\\posts.json",true);
                      file.append(obj.toJSONString()+"\n");
                      file.flush();
                      file.close();

           } catch (IOException e) {}
          list1.clear();
        }
    }
         catch(Exception ex){}
        }

}
```

```
public class DatasetMySQL {
    public static void main(String[] args) throws IOException {
       createPosts();
       createComments();
       createTags();
       createCategories();
       createPostTag();
       createPostCategory();
    }
    static String[] tags = {"Programming", "MongoDB", "Databases",
"Linux", "Windows", "Hacking", "VMWare", "VOIP", "CPU", "Harddisk"};
    static String[] categories = {"Development", "Gaming", "Secu-
rity", "Operating system", "Mobile devices"};
    static Random r =new Random();

    private static void createPosts() throws IOException
    {
        ArrayList names = new ArrayList();
        try(BufferedReader br = new BufferedReader(new File-
Reader("D:\\ DataSetNames.txt"))) {
        String line = br.readLine();
        while (line != null)
        {
            names.add(line);
            line = br.readLine();
        }
        }
        String text = "Lorem Ipsum is simply dummy text of the print-
ing and typesetting industry. Lorem Ipsum has been the industry's
standard dummy text ever since the 1500s, when an unknown printer
took a galley of type and scrambled it to make a type specimen book";
        FileWriter fw = new FileWriter("D:\\Posts.csv");

        for(int i=1; i<=10000000; i++)
        {
            fw.append(i+";");
            fw.append(i+". post;");
            fw.append(text.substring(r.nextInt(text.length()))+";");
            fw.append((CharSequence) names.get(r.nex-
tInt(names.size()))+";");
            fw.append(r.nextInt(300)+";");
            fw.append("201"+r.nextInt(5)+"-0"+(r.nextInt(8)+1)+"-
"+r.nextInt(2)+""+(r.nextInt(8)+1)+" "+r.nextInt(1)+ r.nex-
tInt(9)+":"+r.nextInt(5)+r.nextInt(9)+":"+r.nextInt(5)+r.nex-
tInt(9)+"\n");
        }
        fw.flush();
        fw.close();
    }

    private static void createComments() throws IOException
    {
        ArrayList names = new ArrayList();
        try(BufferedReader br = new BufferedReader(new FileReader("D:
\\DataSetNames.txt"))) {
        String line = br.readLine();
        while (line != null)
        {
            names.add(line);
            line = br.readLine();
    }
        }
        String text = "Lorem Ipsum is simply dummy text of the print-
ing and typesetting industry. Lorem Ipsum has been the industry's
standard dummy text ever since the 1500s, when an unknown printer
took a galley of type and scrambled it to make a type specimen book";
        FileWriter fw = new FileWriter("D:\\Comments7.csv");

        for(int i=1; i<=100000000; i++)
        {
             fw.append(text.substring(r.nextInt(text.length()))+";");
             fw.append((CharSequence) names.get(r.nex-
tInt(names.size()))+";");
             fw.append("201"+r.nextInt(5)+"-0"+(r.nextInt(8)+1)+"-
"+r.nextInt(2)+""+(r.nextInt(8)+1)+" "+r.nextInt(1)+ r.nex-
tInt(9)+":"+r.nextInt(5)+r.nextInt(9)+":"+r.nextInt(5)+r.nex-
tInt(9)+";");
             fw.append((r.nextInt(9999999)+1)+ "\n");
        }
        fw.flush();
        fw.close();
    }
       private static void createTags() throws IOException
        {
        try (FileWriter fw = new FileWriter("D:\\Tags.csv")) {
            for(int i=0; i<tags.length;i++)
            {
                fw.append(i+";");
                fw.append(tags[i]+"\n");
            }
            fw.flush();
            fw.close();
        }
        }

       private static void createCategories() throws IOException
        {
        try (FileWriter fw = new FileWriter("D:\\Categories.csv")) {
            for(int i=0; i<categories.length;i++)
            {
                fw.append(i+";");
                fw.append(categories[i]+"\n");
            }
            fw.flush();
            fw.close();
        }
        }

       private static void createPostTag() throws IOException
       {
            try (FileWriter fw = new FileWriter("D:\\PostTag.csv")) {
            for(int i=1; i<=10000000;i++)
            {
                int count= r.nextInt(tags.length);
                Set<Integer> tag = new HashSet();
                for(int j=0; j<count; j++)
                {
                    tag.add(r.nextInt(tags.length));
                }
                for(int t:tag)
                {
                    fw.append(t+";");
                    fw.append(i+"\n");
                }
            }
            fw.flush();
            fw.close();
            }
        }

         private static void createPostCategory() throws IOException
        {
            try (FileWriter fw = new FileWriter("D:\\PostCate-
gory.csv")) {
            for(int i=1; i<=10000000;i++)
            {
                int count= r.nextInt(categories.length);
                Set<Integer> category = new HashSet();
                for(int j=0; j<count; j++)
                {
                    category.add(r.nextInt(categories.length));
                }
                for(int t:category)
                {
                    fw.append(i+";");
                    fw.append(t+"\n");
                }
            }
            fw.flush();
            fw.close();
            }
        }
}
```