UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

POST GRADUATE STUDIES- SECOND CYCLE

THESIS TITLE:

Analysis and comparison of NoSQL Databases to Relational Databases

Supervisor:

Asst. Prof. Dr. Xhemal Zenuni

Candidate:

Raif Deari

Tetovo, 2018

Declaration of originality

I hereby confirm that this thesis is my own work and that I have not sought or used inadmissible help of third parties to produce this work and that I have clearly referenced all sources used in the work. I have fully referenced and used inverted commas for all text directly or indirectly quoted from a source. This work has not yet been submitted to another examination institution.

Raif Deari

Proofreading declaration

I hereby confirm that this thesis has been correctly proofread and that it meets all the linguistic requirements prior to the publication phase.

In Tetovo, on 11.07.2018

Proofreader: Kujtim Ramadani

Abstract

The aim of this master thesis is to make a comprehensive analysis and comparison between NoSQL and relational databases. We review and evaluate data storage and data management principles of each type of concerned databases. In addition, we evaluate the performance of CRUD operations using different scenarios on MongoDB and MySQL as representatives of two respected data models. The results give insights to advantages and disadvantages of each database model.

Contents

List of tables

List of Figures

Introduction

NoSQL Databases are a new generation of databases. They are mostly non-relational and mainly developed for applications, which require data management where relational databases do not fulfill the requirements.

These so-called new generation databases are usually referred to as NoSQL (Not-Only-SQL) and in the subsequent text, the NoSQL acronym will be used to refer to this category of databases.

NoSQL usually refers to databases, which meet the needs of modern developments in business and information technology. These new developments need scaling to previously never-thought levels while remaining always available with a respected speed (Datastax, 2017). The same source states that “By all accounts, the consensus of IT professionals and industry database experts seems to be that NoSQL is here to stay”.

Apart from that, NoSQL meets the needs of modern business in timely fashion; it provides a very flexible data model, being horizontally scalable while also supporting a distributed architecture. However, there are two main questions, which should be answered in the case of NoSQL: When and why should NoSQL Databases be used? - These questions are very important since they help us decide whether a relational solution is sufficient for an application or NoSQL should be used instead.

In (Datastax, 2017), authors state that among the main reasons why NoSQL is finding an ever-growing use among modern day businesses is that of Big Data which is becoming a synonym to NoSQL thus being its main “advocate”.

According to “ (Storey & Song, 2017)” big data refers to large amounts of data captured by organizations in a structured or unstructured format so that data-driven analysis, decisions or actionable insights can be obtained.

Big Data as a concept is older than NoSQL Technologies since it mainly refers to the traditional enterprise data, which mainly includes customer information and transactions.

Other types of Big Data include machine-generated data and “social data”. Machine-generated data include streams of information from sensors, logs, equipment and other while Social data include customer feedback, blogging sites such as Twitter or social media platforms; these data are user-generated and mainly unstructured. The main characteristics of Big Data as explained by (Datastax, 2017) include volume, velocity, variety, and value.

Volume is the amount of data, which is generated, and it is in larger amounts from traditional data. In (Datastax, 2017) authors explain it by taking the example of a single jet engine which generates 10 terabytes of data in 30 minutes of its operation; with 25,000 flights per day the amount of data reaches petabytes.

Velocity is the frequency of data and it is characteristic of social media sites whose data is mainly streams of characters where the data is transmitted with high frequency, thus making an important characteristic of big data.

Variety is another important characteristic of Big Data and it is the main point of differentiation from traditional data and one of the more important reasons why Big Data and NoSQL have almost become synonyms of each other.

Traditional structured data tends to be relatively well defined and has a schema, which changes either slowly or never. While in contrast to it, non- traditional data formats influenced by the digital- economy trends represent a fast- changing ever-growing type of data, which goes well with the ability of NoSQL to handle unstructured data.

The last attribute of Big Data is the economic value. This last attribute incorporates the first three since not all the data collected may be economically valuable; however, the big amount collected in high velocity with a high variety makes possible to identify economically valuable data.

In this thesis, the main point of focus will be the analysis of differences between NoSQL Databases Relational Databases, while other points of interest will be the types of NoSQL Databases and their ability to fulfill the expectations in scenarios where NoSQL Databases can be used.

Nowadays huge amounts of data are generated daily. This data comes from the ever-growing and data intensive way of how organizations are set up. Management of this data was done entirely using Relational Databases which provide a centralized system of control, redundancy control and inconsistency elimination however with drawbacks and limitations such as performance, ease of maintenance or system scalability of which the last one has become a primary factor for looking at other data storage alternatives such as NoSQL Databases.

NoSQL Databases mostly start as development of practitioners looking fit special requirements when faced with the above-mentioned limitations while nowadays these databases can be seen as replacement of relational databases.

To better understand the differences of these two types of databases the following sections gives a briefing on Relational Databases, their features, and their approach towards managing data while making analogies with NoSQL Databases.

Relational Databases are described as databases based on the relational model introduced by F. Codd, working with a support for SQL (Standard Query Language) (Rafique, 2013). The main feature of Relational Databases is the support of a strong and strictly regulated schema built on basis of: storage of data in tables (rows and columns), usage of keys and data integrity constrains.

Rows, Columns and Tables

The relational database model represents data with tables, known as relations. Every relation has a certain number of attributes, represented by columns of the table, where every column has its own atomic type such as an integer or string.

ID	Name	Age

Figure 1, Row in Relational Databases

Name
John
Alex
Maria

Figure 2, Column

ID	Name	Age
1	John	23
2	Alex	22
3	Maria	24

Figure 3, Table in Relational Databases

Rows of the table are called *tuples*; every tuple has its components, which must belong to the same atomic type as its attribute.

Constraints

Constraints are database integrity rules to maintain database consistency. Constraints are used to limit the type of data, which can be inserted into a table, this ensures data accuracy and reliability. Constraints can be implemented in different levels (column level or table level), some of the most commonly used constraints are:

- Not Null Constraint, which ensures that columns do not have a null value.
- Default Constraint, which ensures that a default value is inserted when the user specifies none
- Unique Constraint, which provides with a guarantee that all the values are different
- Primary Key, which uniquely identified each row in a table.

ACID- Atomicity, Consistency, Isolation and Durability

One of the most important concepts in Relational Databases is ACID, which are a set of properties that a system must maintain to achieve and ensure accuracy, completeness and data integrity.

Atomicity is an attribute of relational databases, which states that a transaction should either go through and execute completely or not execute at all. If one part of the transaction fails to complete, it fails completely. Atomicity treats every transaction as an atomic unit, and states that there should not be a state in the database during the transaction, which is left partially complete.

Consistency requires that the database remains in a consistent state after any completed transaction, and there should not be any adverse effect on the data residing in the database. If the database has or is in a consistent state before the transaction, it should remain so after it.

The virtue of isolation means that every single transaction is executed independently of the other transactions regardless of other conditions such as accessing the same database at the exact same time. Therefore, no transaction affects the existence of any other transaction.

Durability is a property, which ensures that the database is durable enough to hold all its latest transactions even if the system fails or restarts. When a transaction completes and changes some certain data in a database and commits, then the database will hold the modified data.

ACID alternatives in NoSQL- Defining BASE and CAP Theorem

The ACID Model, which databases need to fulfill to guarantee Data Atomicity, Consistency, Integrity, and Durability, is a very important model, which is followed by all Relational Databases and some NoSQL ones.

To fulfill ACID properties, NoSQL Databases use another process described in (Pritchett, 2008), as ACID divided in two steps or a two phase commit (2PC), where each transaction is pre-commit and waiting for a commit allowance and if this allowance is achieved then the commit goes through. If for any reason, any of the databases involved with the action vetoes the commit than all databases are required to roll back the transaction.

Apart from ACID, there are other approaches, which try to achieve the same properties ensured by ACID, but with a different mindset, one of them being BASE (Basically Available, Soft State, Eventually Consistent).

Dan Pritchett (Pritchett, 2008) in his publication in ACM Magazine states that: “BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads to levels of scalability that cannot be obtained with ACID”.

Data Availability is achieved through support for partial failure, since NoSQL is distributed and there is no single point of failure. If one of the databases fails you must deal with only a percentage of users not being able to complete transactions.

The table below shows conceptual differences between the Relational and NoSQL approaches towards the same goal (Chandra, 2015).

ACID (Relational Databases)	BASE (NoSQL)
Strong consistency	Weak consistency
Isolation	Last write is saved
Robust database	Simple database
SQL Support	Distinctive for different types
Available and consistent	Available and partition- tolerant
Scale- up	Scale- out
Shared	Parallel

Table 1, ACID vs. BASE Properties

Another approach is CAP Theorem (Consistency, Availability, and Partition Tolerance). Eric Brewer on his paper "CAP Twelve Years Later: How the "Rules" Have Changed" on CAP Theorem states that only two of the three can be supported and guarantee. In the cases where we have horizontal scalability strategies, developers are forced to choose between consistency and availability (Brewer, 2012).

The above-explained concepts are a basic package of concepts for Relational Databases and their key characteristics while the following section introduces to some key differences between them (Relational Databases) and NoSQL Databases.

The foremost essential difference between the two technologies is that Relational Databases have a structured and well- organized approach to data management, while NoSQL goes by the unstructured approach. This comes because of the fact that relational databases were built at a time where data was fairly structured and defined by their relationships, in the other hand NoSQL is especially designed to handle unstructured data, which makes up the majority of the data, which exists nowadays.

Another significant difference between these two database models is the ACID compliancy, which is strictly implemented in Relational Databases while the implementation in NoSQL Databases is less-to-none.

The data model is another difference of the two systems. In the case of Relational Databases, the data model is pre-defined and strict on the requirements, which should be fulfilled. NoSQL Databases in the other hand, have a less organized data model since NoSQL at its creation is described as a schema-less system. Another factor contributing to the difference in the data model is the wide range of NoSQL Databases where different data models are developed and implemented for different use cases.

The following chapter will give a better clarification on these differences since in it we explain the different divisions of NoSQL Databases, while also introducing different database “vendors” in each category.

Background and Related Work

Relational Databases or the relational model is a single model without divisions as we will see later in this chapter for NoSQL Databases, however Relational Database Management Systems between them have small changes on their approach towards managing data. They all support a standard query language (SQL) and all must be ACID compliant.

NoSQL on the other hand is different since there are a huge variety of technologies within the concept of NoSQL (Bugiotti & Cabibbo, 2013). This variation comes because of the nature of NoSQL, which as mentioned in the first chapter "started as a development of practitioners for specific needs".

The following categorization of NoSQL Databases is the one, which comes up more often across studies (Cattell, 2010) (Bugiotti & Cabibbo, 2013):

- Key- Value Databases
- Document Databases
- Column- based Databases
- Graph Databases (not discussed in this thesis)

The following section introduces these categories. We also look at the representatives from each of these categories such as Oracle and Redis for Key-Value Databases, Mongo DB for Document Databases and HBase and Cassandra for Column-based Databases.

Key-Value Databases (Redis and Oracle)

Key- Value Databases in simple terms are NoSQL Databases, which in their essence have the ability to store data, called a value inside a key. This data can later be retrieved only if we know the exact key used to store it (Redis, 2017). Another more detailed definition of Key-Value Databases is "a non-relational database design or data structure that maps from arbitrary names (keys) to arbitrary objects" (Beltrame, 2013).

Key- Value Databases allow the application developer to store schema-less data. This data usually consists of a string, which represents the key and the actual data, which is considered to

be the value in the “key-value” relationship. The data itself is usually some kind of primitive data type found in programming languages (a string, an integer an array) or an object that is being marshaled by the programming languages bindings to the key value store. This replaces the need for fixed data model and makes the requirement for property-formatted data less strict (Seeger, 2009).

The following example explains this in a practical way. It is the case where the “pstore” library is used using Ruby.

```
require "pstore"

store= PStore.new("data-file.pstore")
store.transaction do #begining of transaction

    store[:single_object] = "Lorem ipsum dolor sit amet..."
    store[:hierarchy_object]= {"Marc Seeger" => ["ruby", "mysql"]

                               "Rainer Wahnsinn" => "php", "nosql"}
end
```

Figure 4, Ruby with NoSQL

In the above example, we see a key value store called “data-file.pstore” where we have added two objects (single_object and hierarchy_object). The first one is a simple object or a string, while the second one is of a more complex data type. It is a hash, which contains arrays of strings. Even though the syntax differs across key-value databases, they all work with similar concepts.

Now that the general concept of Key- Value Databases is explained, some top database systems of this category are reviewed. In this category, Redis, Oracle (Key- Value), and Amazon’s Dynamo DB are amongst most popular.

Redis

Redis as described by the developers (Redis, 2017) is a “very simple database”, which implements a dictionary where keys are associated with values. The interesting thing about

Redis is that keys can be associated to values which are not only string, but they can be lists or sets with many server- side operations associated to this data types.

When it comes to data storage, Redis takes the whole dataset in memory while dumping it on disk asynchronously from time to time. The developers describe this asynchronous dump to a user specific desire since it can be set after many changes are made or after an amount of time.

Redis also supports the master- slave replication to achieve maximum consistency since the asynchronous data save becomes an issue if a system crash occurs. Issues or drawbacks which Redis can have is the amount of RAM it requires since it only dumps the data when told so.

The usage of an asynchronous saving model can also become a deal breaker when we deal with sensitive data as in cases of banks where the reliability should never be compromised even in cases when the server goes through states such as a power loss or other possible technical faults.

This drawback in cases of high data sensitive data becomes an advantage in the other side of the spectrum when we deal with “not-so-important” data, since Redis is a very high-performing platform in this case.

In today’s web, the biggest data generators are social media platforms, which fall in the category of “not-so-important” data since it is never a big deal if the last status update is not saved or a tag in a single picture is not there, while it is of vital importance to deliver high speeds to the user without the backend slowing down due to unneeded transactional integrity.

Redis can perform over 100,000 operations (write, read, increment) per second with a normal Linux box with 50 concurrent clients (Seeger, 2009).

As of November 2017, Redis is ranked as the top Key- Value Database while it is ninth overall among all the other types of Relational and NoSQL Databases with a benchmarking score of 121.18 in DB Engines Ranking (DB-Engines, 2017).

Oracle NoSQL

Oracle being the leader it is on Relational Databases has its own NoSQL System and it is called Oracle NoSQL.

Oracle's NoSQL database is based on Berkeley DB and it comes as part of what Oracle calls the "Oracle Big Data Appliance", which comes as a Big Data Platform which according to (Oracle, 2013) has three main infrastructural requirements: Acquisition, Optimization and Analysis of Big Data.

- Acquisition of Big Data

Acquisition phase of Big Data has completely changed from the days before NoSQL. Since big data refers to streams of data with high velocity and variety, the infrastructure required to support the acquisition of big data must deliver:

- Low and predictable latency
- Handle high transaction volumes (mostly in distributed systems) and
- Support dynamic data structures.

Nowadays NoSQL databases are used for acquiring and storing big data since they are very well suited for dynamic data structure and are highly scalable systems.

A good example on how NoSQL works in accord with the infrastructural requirement of acquisition is the example of social media, which may often change as applications; however, the underlying structure does not change and is kept simple usually with the key-value mindset where a key point identifies the data point, and then a content container holds the relevant data.

- Organization of Big Data

Organizing data or in classical terms Data Integration, is a process in which data warehouse try to organize data to its initial destination location to save time and money by not moving large volumes of data.

Oracle describes this quality as: “The infrastructure required for organizing big data must be able to process and manipulate data in the original storage location; support very high throughput (often in batch) to deal with large data processing steps; and handle a large variety of data formats, from unstructured to structured” (Oracle, 2013).

- Analysis of Big Data

During the organization phase, data is often moved or distributed, thus the process of analysis has to have the ability of handling data analysis in distributed environments.

For analyzing Big Data, the infrastructure should also be able to handle deeper analytics such as statistical analysis, or data mining on different data types, stored in different systems.

Other important features, which must be supported, are the ability to scale to very large data volumes, keeping short response times while also being able to make automated decisions based on the given analytical models.

Oracle combines all these requirements to create a system of services, which incorporates big data with its own NoSQL Database (Oracle NoSQL) and technologies such as Hadoop. This makes possible a complete integrated solution to address the full spectrum of enterprise big data requirements.

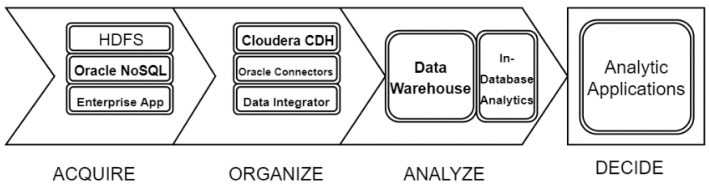

Figure 5, NoSQL structural requirements (Oracle, 2013)

Oracle has created a big data appliance, which is an engineered system that combines optimized hardware with software to deliver complete and easy-to-deploy solution for data

storing purposes. The so-called Big Data Appliance of Oracle combines open source software with specialized software to address enterprise requirements as seen on figure 6.

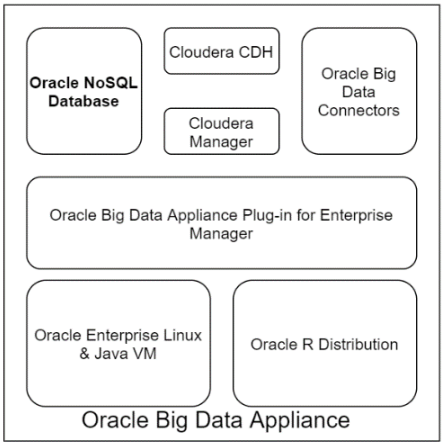

Figure 6, Oracle Big Data Appliance (Oracle, 2013)

In their paper "Oracle: Big Data for the Enterprise" they describe it as "a general-purpose database, enterprise class key value store adding an intelligent driver on top of distributed Barkeley DB" and that this added intelligent driver "keeps track of the underlying topology, shards the data, and knows where data can be placed with the lowest latency".

Oracle NoSQL differs from many other NoSQL databases in the fact that it is easier to set up, configure and manage, it also supports a wide set of workloads and delivers what they call enterprise- class reliability. The architecture of Oracle NoSQL Database looks as below:

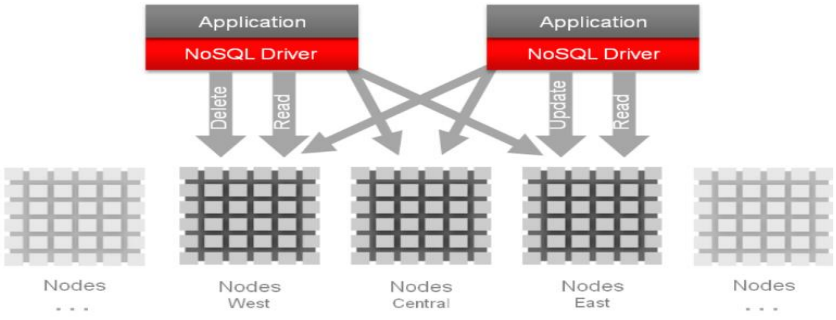

Figure 7, Oracle NoSQL Architecture (Oracle, 2013)

The primary use cases for Oracle NoSQL are low latency data capture and fast querying of the data typically by key lookup, as of November 2017 (DB-Engines, 2017), Oracle NoSQL has a

score of 2.78 and stands as the 12th key- value database while it is 77th in the overall ranking always by DB- Engines rating.

Document Databases (Mongo DB and Couch DB)

Document Databases are NoSQL databases, which use documents to store data. Documents in Document Databases are structured depending on the implementation, but their usual structure comes in a form of XML (Extensible Markup Language), JSON (JavaScript Object Notation) or YAML (YAML Ain't Markup Language).

They all have in common the semi- structured nature meaning that the document does not have to conform to any static schemas nor tables; instead, they use tags and other methods that allow related documents to contain different keys and values (Omji, et al., 2018).

In some types of Document Databases, usage of references like the ones used in SQL or Relational Databases is possible (mishra2018).

Two of the most popular NoSQL Document Databases are Mongo DB and Couch DB, databases that we will discuss in the following section.

Mongo DB

Mongo DB is a document- oriented NoSQL Database written in C++ and developed by 10gen. From their website, we can understand that Mongo DB is focused on the ease of use, performance and high- scalability.

Mongo DB is organized in documents which use the binary form of JSON called BSON or Binary JSON. When users enter data, in Mongo DB they use JSON, which is then converted to BSON, and when at the other end this data is retrieved another conversion from BSON to JSON happens, meaning that BSON is only used for internal purposes and the user never faces it.

A JSON document is one or more key- value pairs and a Mongo Document is simply a JSON document.

Since Mongo DB uses JSON, it is schema-less, which means that there is no grouping of documents that share the same keys as in the relational model where the relation roughly fills this purpose.

Instead, similar documents, which contain data about the same thing but with different key value pairs, are grouped together in what is called a collection.

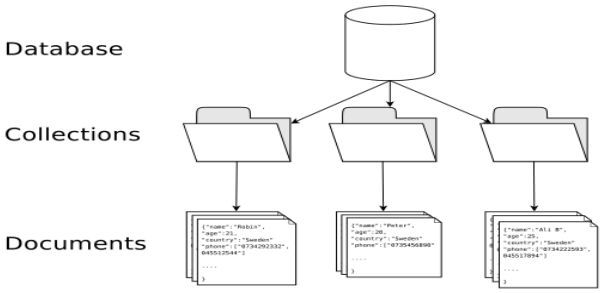

Figure 8, Different Key- Value groupings

An important characteristic of Mongo DB is its support to indexing any attributes of a document similar to how Relational Databases offer indexing on any column. Indexes in Mongo DB are implemented as B- Trees which if rightly created can give a dramatic change in performance, especially in querying.

Like many other NoSQL Databases Mongo DB also supports replication such as Single Master/ Single Slave, Multiple Masters/ Multiple Slaves, Master/Master etc. (Schmitt & Majchrzak, 2012) (Bhardwaj, 2017) (Henricsson, 2011).

As of November 2017, Mongo DB ranks 5th on the overall database ranking of db- engines while it is the top ranked document- oriented database with a score of 330,47 (DB-Engines, 2017).

Couch DB

Couch DB is another very important document- oriented database; it is developed in Erlang by the Apache Software Foundation.

What makes Couch DB special is its RESTful API, which lets any environment that allows HTTP requests to access data from a database set up in Couch DB.

In terms of data storage, it is similar to Mongo DB since it stores data as Binary JSON while it lacks collections, so the documents are directly found in the database, where each document has a unique id, which can be assigned manually when inserting documents, or automatically by Couch DB.

There is no restriction in terms of the number of key- value pairs for documents and there are no size restrictions as well.

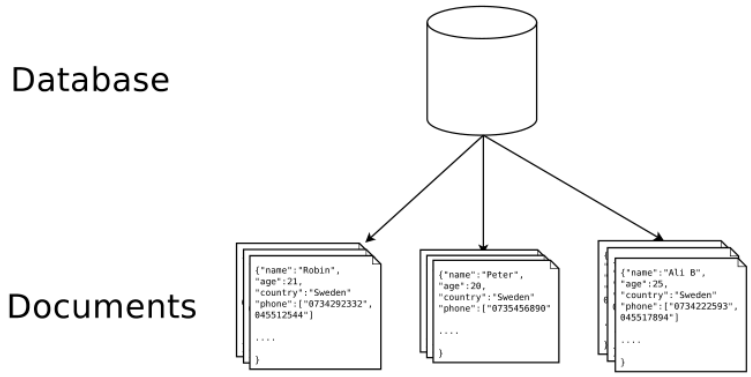

Figure 9, CouchDB storage type

Relational databases typically use static data and dynamic queries; schemas are fixed, and SQL queries are dynamic. Couch DB, however, has turned this upside down. Since it uses JSON documents, the data is dynamic. Querying data in Couch DB is done through views.

There are two kinds of views:

- Permanent Views- Static
- Temporary Views- Can be provided ad-hoc.

Views show the results of Map/Reduce functions. The user writes map functions and iterate over all documents in the database to check if the documents match the criteria specified in the function by the user. If everything matches, and a result is hence found, the document (or selected parts of it) are emitted using the emit function (Henricsson, 2011).

As of November 2017, Couch DB is ranked 28th overall with a 20.51 score, while it is 4th on document stores only (DB-Engines, 2017).

Column- based Databases (Cassandra and HBase)

Column- based Databases also known as Wide- Column or Column- Family Databases are alike Document Databases which employ a distributed, column- oriented data structure that accommodates multiple attributes per key (Moniruzzaman & Hossain, 2013).

Extensible Record Databases are in a way like Key- Value databases since a lot of them have what (Moniruzzaman & Hossain, 2013) calls "Key- Value DNA" such as Dynamo DB and Cassandra while other are inspired and built on Google's Bigtable.

Google's Bigtable as described by Google is "Google's NoSQL Big Data database service. It is the same database that powers many core Google services, including Search, Analytics, Maps, and Gmail. Bigtable is designed to handle massive workloads at consistent low latency and high throughput, so it's a great choice for both operational and analytical applications, including IoT, user analytics, and financial data analysis."

Another simplified definition of Extensible Record Stores is a database with basic data model using rows and columns while their basic scalability model is splitting both rows and columns over multiple nodes (Abadi, 2007).

As seen from the above definitions we face two basic ingredients: Rows and Columns.

- Rows

Rows are split across nodes through sharding on the primary key. They usually split by range rather than function; this strategy makes possible for queries not to have to go to each node.

- Columns

Columns of a table are distributed over multiple nodes using the concept of column groups, which is a way for the customer to decide which columns are best stored together.

With these two concepts we see that we have not only horizontal but vertical partitioning also, and the good thing about Column- based Databases is that these partitioning methods can be both used at the same time.

A good example is given by (Abadi, 2007): “If a customer table is partitioned into three column groups (say, separating the customer name/address from financial and login information), then each of the three column groups is treated as a separate table for the purposes of sharding the rows by customer ID: the column groups for one customer may or may not be on the same server. The column groups must be pre-defined with the extensible record stores.”

However, that is not a big constraint, as new attributes can be defined at any time. Rows are analogous to documents: they can have a variable number of attributes (fields), the attribute names must be unique, rows are grouped into collections (tables), and an individual row’s attributes can be of any type”.

Primary uses of Column- based Databases are:

- Distributed Data Storage
- Large- scale, batch- oriented data processing
- Exploratory and predictive analytics

Some advantages of these NoSQL Database according to (Cattell, 2010) are:

- Improved bandwidth utilization

In a Column- based Databases only the data which is accessed by the query needs to be read off the disk, while in other types surrounding attributes need to also be accessed.

- Improved data compression

Storing data from the same attribute domain together increases locality and thus data compression rate.

While disadvantages of Column- based Databases are:

- Increased disk seek time

Disk seeks in-between each read of a block sometimes are needed since multiple columns are read in parallel, even though in large applications this cost is kept small using large disk pre-fetches.

- Increased cost of inserts

Insert queries are one of the main drawbacks of this type of NoSQL Databases since multiple distinct locations on the disk must be updated for each inserted tuple.

The following section looks at the two leading Extensible Record Databases, Cassandra and HBase, which both better define and show the characteristics of this type of NoSQL Databases.

Cassandra

Cassandra supports partitioning and replication, while failure detection and recovery are fully automatic. A drawback of Cassandra when compared with other Extensible Record Stores is that it has a weaker concurrency model because there is no locking mechanism and replicas are updated asynchronously (Abadi, 2007).

Cassandra is written in Java while developed by Apache; it is open source and supported by Data Stax.

Cassandra automatically connects new available nodes to the cluster and uses accrual algorithms to detect node failure while cluster membership is decided with a gossip- style algorithm.

A novelty of Cassandra is that it brings a new concept to the column groups, the super-columns, which are basically a collection of column groups, these column groups are members of column families which are part of databases called key-spaces.

Same as other systems any row can have a combination of column values, while Cassandra uses an ordered hash index, which gets the most out of both hash and b- tree indexes.

As of November 2017, Cassandra ranks first among Wide Column Stores in db- engines ranking while it is 8th overall with a score of 124.21 points.

HBase

HBase is an Apache project. It is written in Java and patterned after Google Bigtable. Characteristics of HBase are that it:

Uses Hadoop distributed file system, which in the case of Bigtable is different from Google's own file system. HBase puts updates in memory and periodically writes them to disk.

Updates which are flushed to the disk go to the end of a data file so that seeks are avoided, files are also periodically compacted, while for recovery purposes if a server crashes updates are also written to a "ahead log file".

Row operations are atomic and occur with row- level locking, while there is optional/ additional support for transaction with wider range, since these operations are atomic they also have concurrency control which aborts them if there is a conflict with other updates.

Partitioning and distribution are transparent, there is no client- side hashing or any fixed key-space as in other systems, while there is multiple master support so that single points of failures are avoided, HBase also supports MapReduce so that a fair and efficient distribution occurs.

HBase's b- trees allow it to have a fast range queries and sorting, in terms of API there are several APIs that support HBase such as: Java API, Thrift API, REST API, JDBC/ODBC.

As of November 2017, HBase ranks second in Database Engines Ranking of Extensible Record Stores while it is ranked as 16th in the overall ranking with a score of 63.56 points (DB-Engines, 2017).

Criteria of Analysis

This chapter is an analysis and comparison of NoSQL Databases and Relational Databases, the comparison is made based on the hypothesis that NoSQL Databases are competitive with Relational Databases and that they provide significant advantages when it comes to Data Modeling.

In this section, we have analyzed and compared NoSQL Databases to Relational ones in the following categories:

- Data Modeling
- Querying
- Performance

The analysis performed are done in each system's as ideal as possible environment, while already available data is used to prove statements.

Data Modeling

Entity Relationship Model

The Entity Relationship Model is the technique of representing data relationships. A key technique in ER Model is the graphical representation of entities and their relationships to each other called the ER Diagram.

Relationships in ER Model can be of three types:

1. One-to-one Relationship

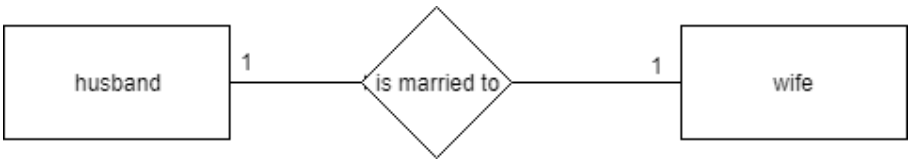

Figure 10, One-to-one relationship

One instance of an entity (husband) is associated with one other instance of another entity (wife). For example, in a database of married couples, each husband is associated with only one wife.

2. One-to-many Relationship

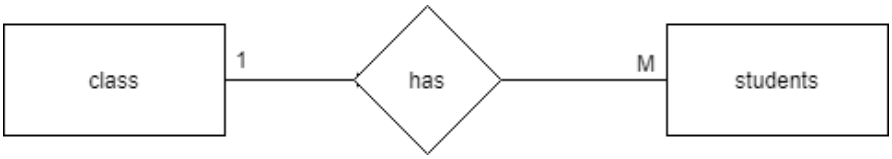

Figure 11, One-to-many relationship

One instance of an entity (A) is associated with zero, one or many instances of another entity (B), but for one instance of entity B there is only one instance of entity A. For example, for a class with all students having their courses in one class, the class name is associated with many different students, but those students all share the same singular association with entity class.

3. Many-to-many Relationship

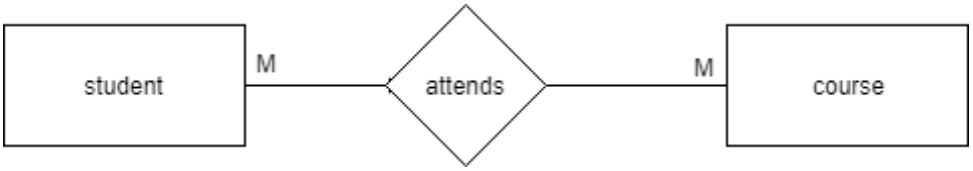

Figure 12, Many-to-many relationship

One instance of an entity (A) is associated with one, zero or many instances of another entity (B), and one instance of entity B is associated with one, zero or many instances of entity A. For example, for a student who attends multiple courses, each instance of a student is associated with many instances of courses, and at the same time, each instance of a course has multiple students associated with it.

Different Data Models for NoSQL Databases

The ER Model is valid in cases where Relational Database model is implemented but not when we deal with NoSQL Databases since their data model varies from a database type to another.,

As mentioned in the second chapter of this thesis we have four main NoSQL Database types however we are working with only three of them: Key- Value Databases, Document Databases and Wide- Column Databases.

DOCUMENT DATABASES	KEY- VALUE DATABASES	WIDE COLUMN STORES
Store data elements in document- like structures that encode information in formats such as JSON.	Use a simple data model that pairs a unique key and its associated value in storing data elements	Also called table-style databases, store data across tables that can have very large numbers of columns.

Table 2, Different types of databases

The main difference between NoSQL Databases and Relational Databases lays in their Data Model. Each NoSQL database type has a different data model.

The following section explains, the approach of each NoSQL Database type (Document, Key-Value and Wide- Column) while comparing them to MySQL as one of the most popular Relational Databases currently in market.

Data Model of Key- Value Stores

Key value stores are like maps or dictionaries where unique keys address data. Since values are uninterrupted byte arrays, which are completely opaque to the system, keys are the only way to retrieve stored data. Values are isolated and independent from each other therefore relationships must be handled in application logic.

Due to this very simple data structure, key value stores are completely schema free. New values of any kind can be added at runtime without conflicting any other stored data and without influencing system availability. The grouping of key value pairs into collection is the only offered possibility to add structure to the data model. Key value stores are useful for simple operations, which are based on key attributes only.

Since most key value stores hold their dataset in memory, they are oftentimes used for caching of more time intensive SQL queries (Funck & Jablonski, 2011).

Data Model of Document Databases

Document Stores encapsulate key value pairs in JSON or JSON like documents. Within documents, keys must be unique. Every document contains a special key “ID”, which is also unique within a collection of documents and therefore identifies a document explicitly.

Document stores offer multi attribute lookups on records, which may have complete different kinds of key value pairs. Therefore, these systems are very convenient in data integration and schema migration tasks.

Most popular use cases are real time analytics logging and the storage layer of small and flexible websites like blogs (Funck & Jablonski, 2011) (Kaur & Rani, 2013).

Like Key- Value stores, Document Stores do not have any schema restrictions. Storing new documents containing any kind of attributes can as easy as adding new attributes to existing documents at runtime.

Their differences with Key- Value Stores lays in the fact that values are not opaque to the system and can be queried as well. Therefore, complex data structures like nested objects can be handled more conveniently.

Storing data in interpretable JSON documents have the additional advantage of supporting data types, which makes document stores very developer-friendly (Kaur & Rani, 2013).

Data Model of Column- oriented Databases

Column oriented stores also known as extensible record stores are almost all inspired by Googles Bigtable, which is a “distributed storage system for managing structured data that is designed to scale to a very large size” (Chang, et al., 2008).

Bigtable is used in many Google projects varying in requirements of high throughput and latency-sensitive data serving. The data model is described as “sparse, distributed, persistent multidimensional sorted map” (Chang, et al., 2008). In this map, an arbitrary number of key value pairs can be stored within rows. Since values cannot be interpreted by the system.

Relationships between datasets and any other data types other than strings are not supported natively.

Columns can be grouped to column families, which is especially important for data organization and partitioning.

Columns and rows can be added very flexibly at runtime, but column families oftentimes must be predefined, which leads to less flexibility than key value stores and document stores offer.

Due to their tablet format, column family stores have a similar graphical representation compared to relational databases.

	Relational Databases	**NoSQL Databases**
Data Model	ER- Entity Relationship Model	Different in different NoSQL Database types
Schema	Predefined Schema	Schema on read (Schema less Approach)
Scalability	Scaling Up (Vertical Scalability)	Scaling Out (Horizontal Scalability)
Consistency	Strong Consistency Required	Eventual Consistency (Strong consistency is not required)

Table 3, RDBMS vs NoSQL Databases

Data Modeling Techniques

NoSQL Data Modeling is a key step on designing NoSQL Data Managing solutions and it often starts from the application- specific queries or simply said it starts from “What questions do I have?” in opposite of relational data modeling which at its basis has “What answers do I have?”.

In most cases, NoSQL modeling requires a deep understanding of data structures and algorithms whilst these are not as important in relational modeling.

Other differences to have in mind when comparing NoSQL and Relational Data Modeling are data duplication and denormalization.

Even though data modeling techniques are basically implementation theories there can be divisions on specific techniques which can be used. Based on multiple studies NoSQL Data Modeling techniques can be divided in three main categories:

1. Conceptual Techniques
2. General Techniques
3. Hierarchy Modeling Techniques

Conceptual Techniques

Conceptual techniques are described as the basic principles of NoSQL Data modeling and they include:

- Denormalization

Denormalization can be defined as the process of copying the same data in multiple documents or tables so that query processing time is optimized, most of the techniques used for NoSQL modeling leverage denormalization in a form.

Denormalization comes with some trade- offs such as data volume to be queried. Using denormalization all the data needed to process a specific query is grouped in one place, which often means that different query flows need the same data, but in different combinations leading to data duplication.

- Aggregates

Major types of NoSQL Databases provide soft schema capabilities.

Soft schema allows formation of classes of entities with more complex internal structures called nested entities; this feature provides two major advantages:

1. Minimization of one-to-many relationships using nested entities thus reducing joins.
2. Masking the differences between business entities and modeling of heterogeneous business entities using one collection of documents or one table.

- Application Side Joins

Application side joins even joins in general are rarely supported in NoSQL, this comes because of the “question-oriented” approach.

Joins in the case of NoSQL Databases are handled in the designing phase as opposed to relational model where joins are handled at the query execution time, we exceptions in cases when we have:

- Many-to-many relationships which are modeled by links and require joins
- Entity internals subject of frequent modifications,

For example, a messaging system can be modeled as a user entity that contains nested message entities, but if messages are often appended it may be better to extract messages as independent entities and join them to the user at query time.

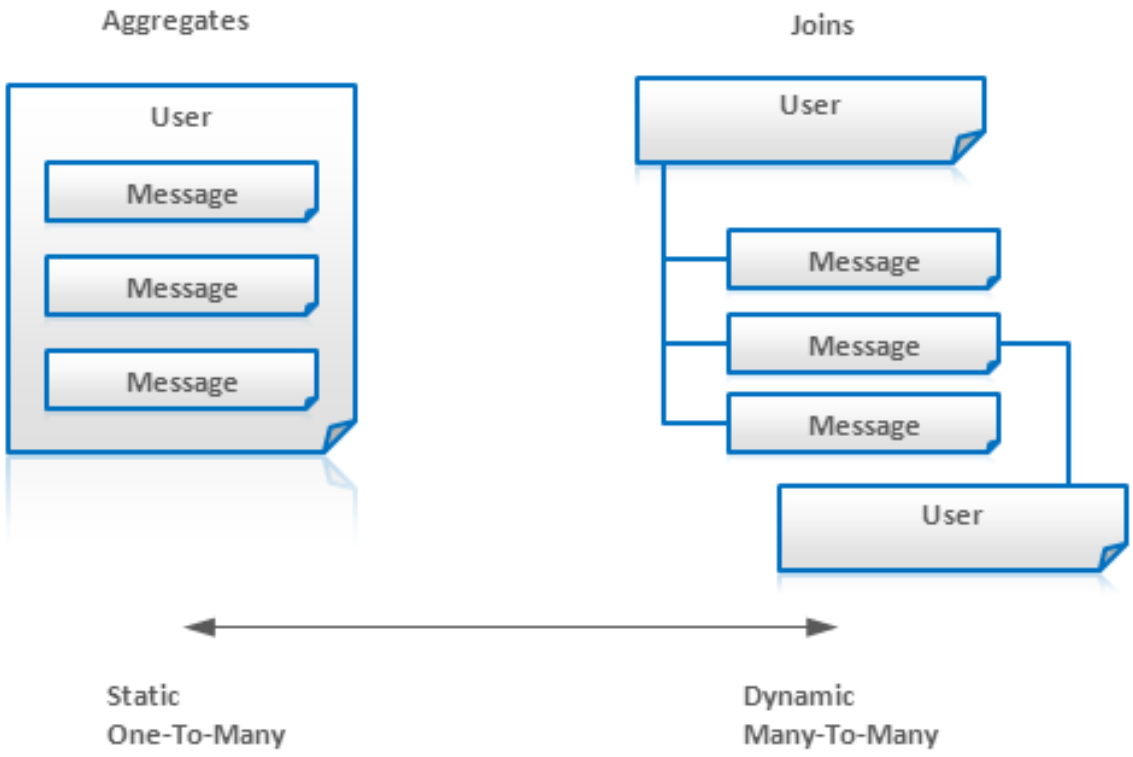

Figure 13, Application Side Joins

General Modeling Techniques

General Modeling Techniques are applicable to a variety of NoSQL Databases and they include the following:

- Atomic Aggregates

Many of NoSQL solutions have limited transaction support, in some cases where this support is missing it can be achieved by distributed locks, and it is common to model data using aggregating techniques to guarantee some of the ACID properties.

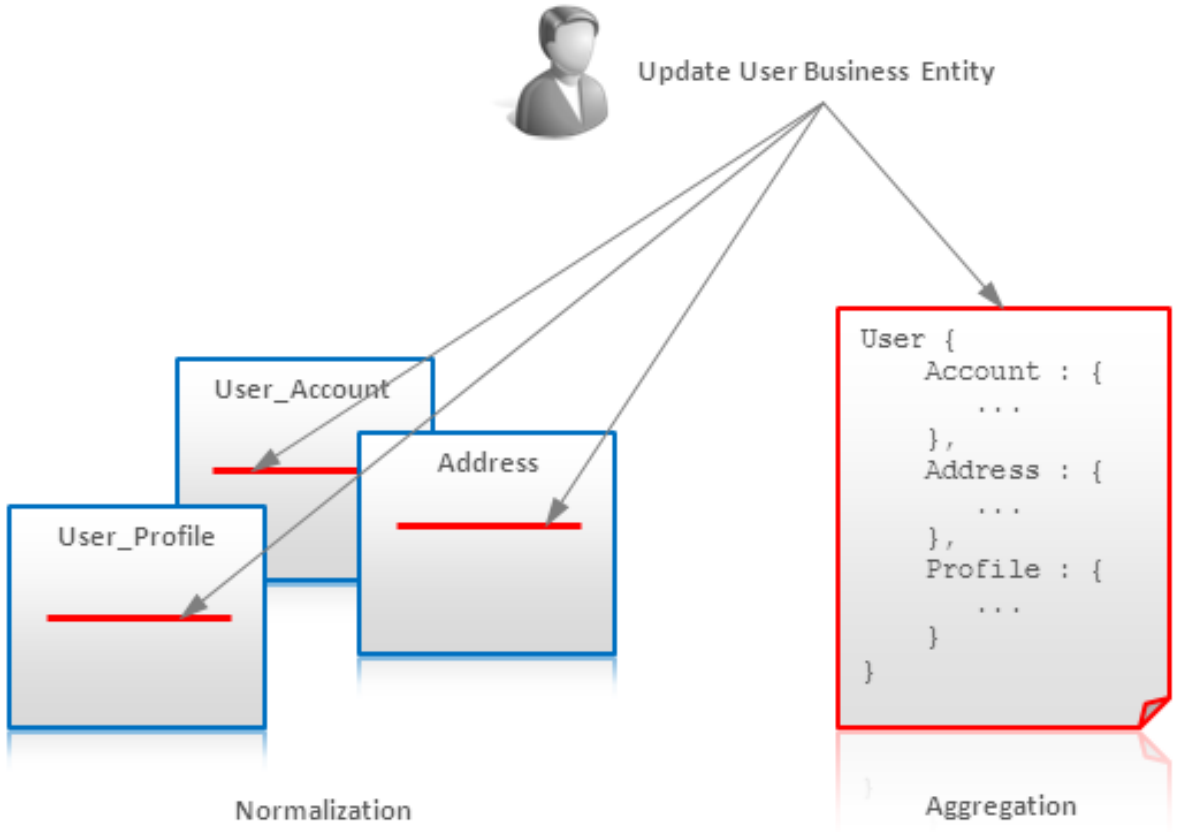

Figure 14, Atomic Aggregation

Atomic Aggregates as a technique to model data is not a complete solution to handle transaction, but if certain guaranties of atomicity are provided by the store, then Atomic Aggregates can be applicable.

- Enumerable Keys

Enumerable Keys technique is a modeling technique applicable only to Key- Value Stores.

Sorting or numbering makes things more complex however it is beneficial if used for certain purposes like modeling for an email application where:

1. Having atomic counters allows generating sequential IDs, so that messages can be stored using userID_messageID as a composite key.
2. In addition, this makes possible the grouping in buckets such as daily buckets for example, allowing the user to traverse a mailbox backwards or forward starting from any specific date.

- Index Table

Index Table is one of the most straightforward techniques that allow taking advantage of indexes in stores that do not support indexes naturally; the most important store that falls in this category is BigTable and all BigTable-style databases.

The idea here is to create and maintain a special table with keys that follow the access pattern.

For example, there is a master table that stores user accounts that can be accessed by user ID. A query that retrieves all users by a specific city can be supported as well by means of an additional table where city is a key.

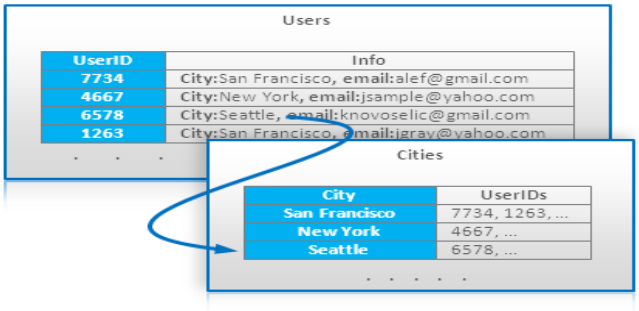

Figure 15, Index Tables

The Index Table can be updated for each update of the master table. Index Tables can be considered as an analog of materialized views in relational databases.

- Composite Key Index

When using stores with ordered keys, Composite Key techniques are very beneficial. When combining composite key with secondary sorting, it is possible to build a kind of multidimensional index.

If we take, for example, a set of records where each record is a user statistic, and if we aggregate these stats by the region the user comes from, we can use keys in a format (Stet:City:UserID) that allow us to iterate over records for a particular state or city if the store supports the selection of key ranges by a partial key match.

```
1  SELECT Values WHERE state="CA:*"
2  SELECT Values WHERE city="CA:San Francisco*"
```

Figure 16, Query example

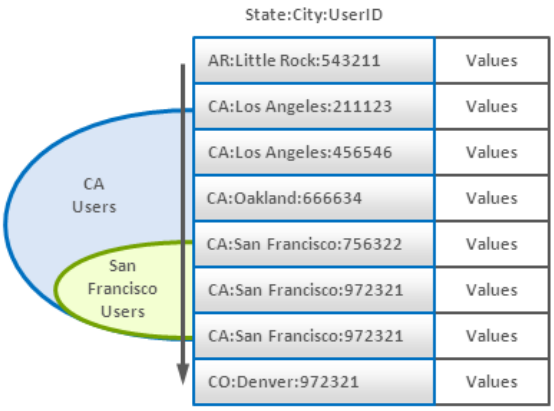

Figure 17, Query Results

- Aggregation with Composite Keys

Composite Keys are used for different types of grouping. If we take, for example, a huge array of log records with information about internet users and their visits from different sites, and our requirement is to count the number of unique users for each site this is like the following SQL query:

```
1 | SELECT count(distinct(user_id)) FROM clicks GROUP BY site
```

Figure 18, Composite key query example

The same situation can be modeled using composite keys with a UserID prefix:

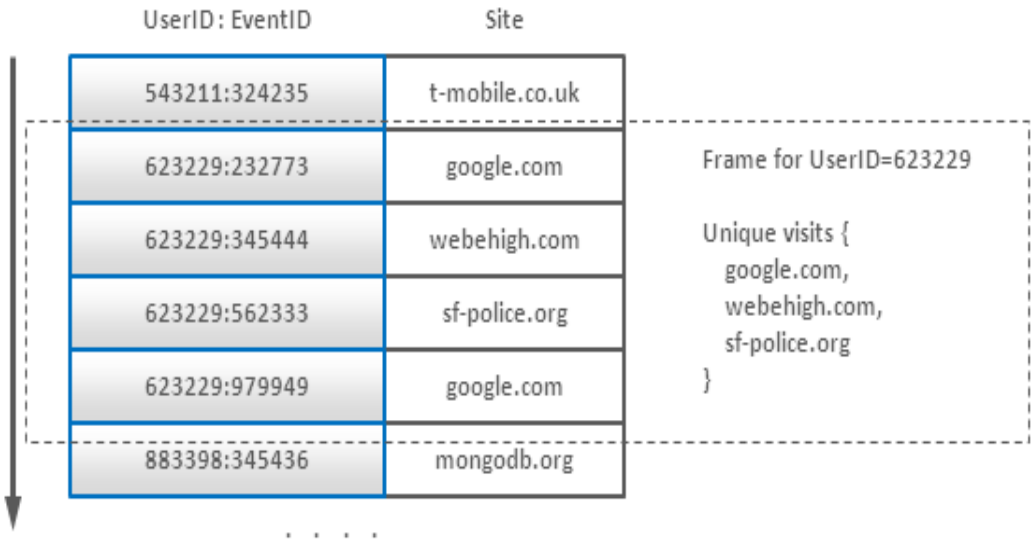

Figure 19, Composite key query results

The idea here is to keep all records for each individual user collocated so that fetching such a frame into memory is possible.

Another alternative technique is having one entry for each user and appending sites to this entry as events in this case visits arrive.

Hierarchy Modeling Techniques

- Tree Aggregation

Trees can be modeled as single record or document. This technique is efficient when a tree is accessed at once, such as the comments of a blog post.

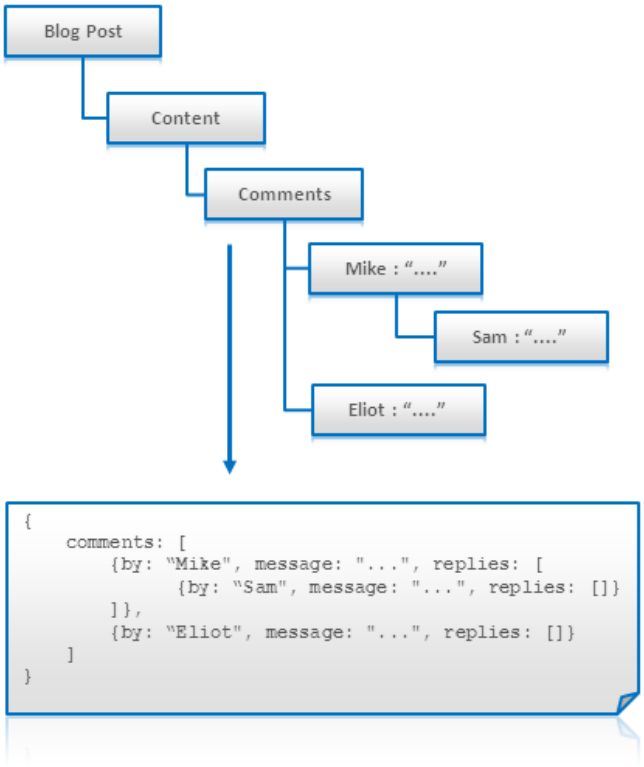

Figure 20, Tree Aggregation

- Adjacency Lists

Adjacency Lists is a technique, which is applicable to Key-Value Stores and Document Databases, and it is a straightforward way of modeling graphs, where each node is modeled as an independent record that contains arrays of direct ancestors or descendants.

- Nested Sets

A standard technique in modeling tree-like structures is Nested Sets. It is mostly used in Relational Databases, but it is also applicable to Key-Value Stores and Document Databases, the idea here is storing leaves of the tree in an array and mapping each non-leaf node to a range of leaves using start and end indexes.

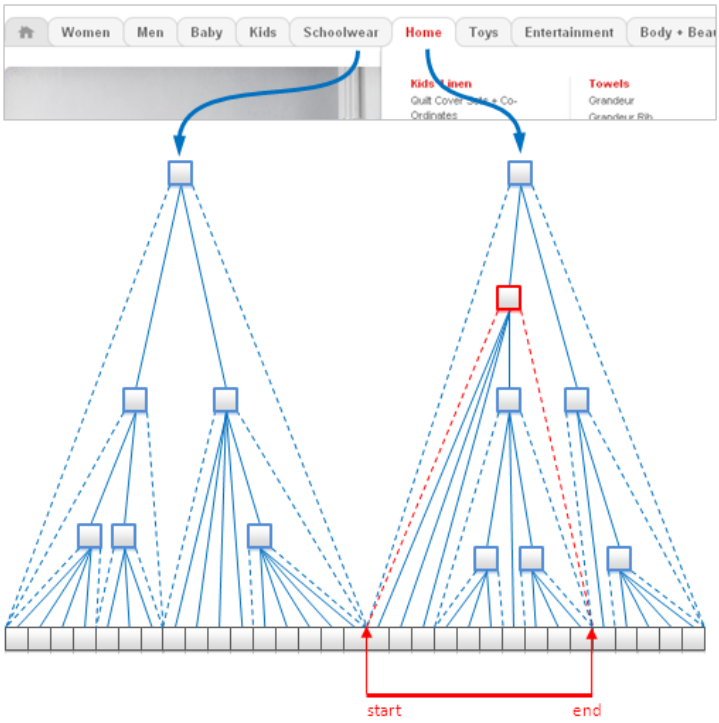

Figure 21, Nested Sets

Such structures are very efficient for immutable data because the memory footprint is very small and allows fetching all leaves for any given node without traversals.

- Nested Documents Flattening: Numbered Field Names

Most Search Engines work with flat documents where each document is a flat list of fields and values, while this works in cases of Search Engines it becomes challenging when mapping business entities for example since the internal structures may be very complex.

One typical challenge of this kind is mapping documents with a hierarchical structure.

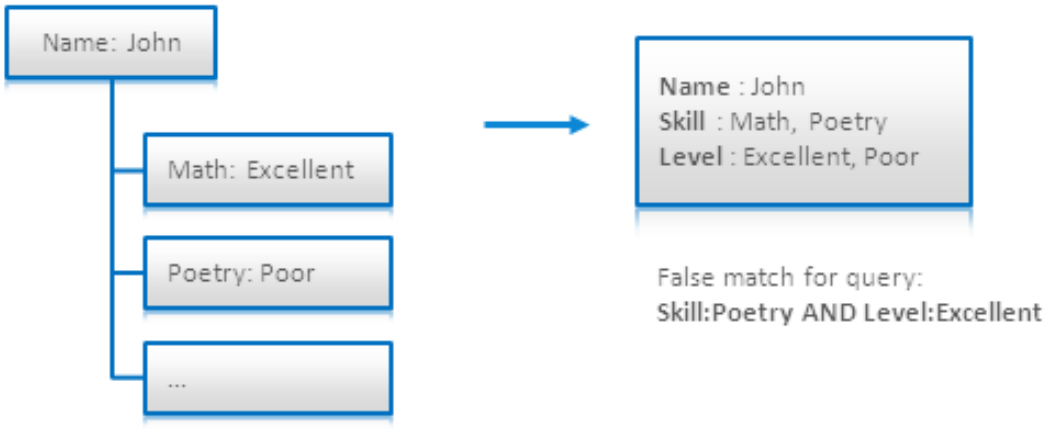

Figure 22, Nested Documents- Numbered fields

In the figure above, we have an example of a business entity, which contains a person's name, and a list of skills with a skill level. An obvious way to model such entities is by creating a plain document with Skill and Level fields, so that a person can be searched by skill or level and everything works fine until we have a combination of these two queries which results in false matches as seen in the figure above.

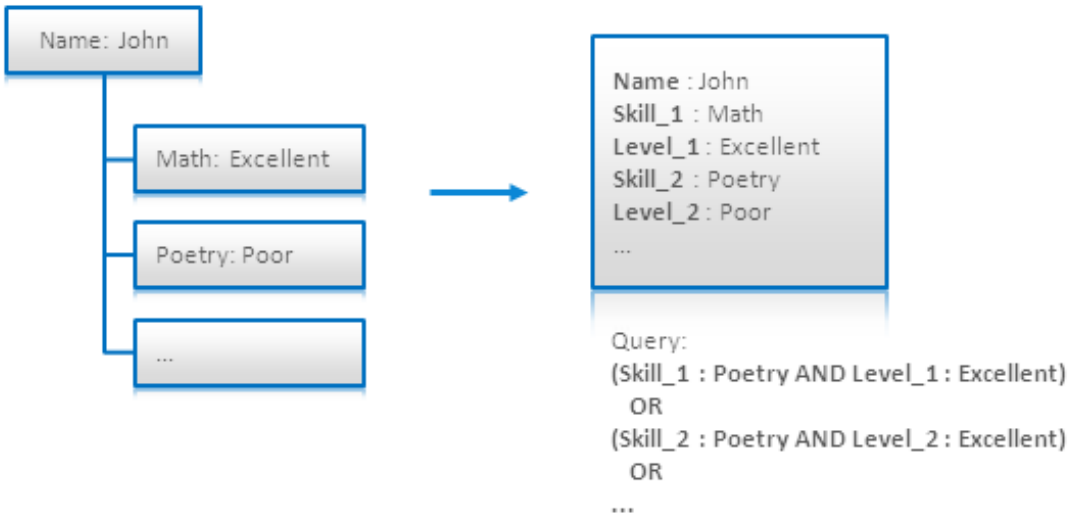

Figure 23, Nested Documents- Numbered fields

One way of overcoming this issue is indexing each skill and the corresponding level as dedicated pairs of fields and then searching for all these pairs simultaneously.

As it can be seen in terms of Data Modeling, there is a huge difference between NoSQL Databases and Relational Database.

Relational Databases use Entity Relational Model as their data model to implement solutions depending on the application, while NoSQL Databases use different data modeling techniques for each database type.

Data Modeling is a very important step which is in most cases is directly related to querying thus being related with performance however in the case of NoSQL systems another important factor which highly contributes to performance is scalability and the way it is achieved.

In the following sections, we will explain scalability to back the hypothesis that NoSQL Databases in general and Document Databases in particular do a better job in this section compared to Relational Databases.

Scalability

Scalability is one of the prominent factors contributing to the advancement of NoSQL Databases. Scalability means partitioning of a system, in this case database, over several nodes; while this is a very important feature to have, it compromises either consistency or availability.

Prioritizing consistency or availability is highly dependent on the nature of the application where the database is implemented; however, there are numerous examples, which trade consistency over availability.

E-commerce applications such as Amazon for example prioritize swift response to users; this forces the system to be highly available for write and read requests, which sometimes requires data to be replicated, and thus forfeiting consistency to an extent.

Another area where NoSQL Databases have found greater implementation is that of social media, an area that in most of the cases prioritizes availability. (Henderson, 2006).

Scalability can also be defined as the border beyond which a system cannot work if expanded; currently scalability can be measured as the ability to handle huge amounts of data while providing uninterrupted service to the applications.

There are three possible kinds of scalability:

1. Vertical Scalability

Vertical Scalability or also known as scaling-up is the addition of more resources (processing power) to the existing machine, as machines run out of capacity more power is added to those machines or they are replaced by newer faster machines, thus increasing the power of individual nodes (power meaning Processing Power, RAM, or Storage Space).

Relational Databases use this type of scaling since their query language (SQL) favors it. One drawback of such approach is that with the increase of power there is an exponential increase in maintenance costs (Henderson, 2006).

2. Horizontal Scalability

Horizontal Scalability also known as scaling- out, is the opposite of vertical scaling since in this case no machine is replaced or upgraded but new machines not necessarily very powerful are added.

Most of NoSQL Databases including Mongo DB, HBase, Dynamo DB are horizontally scalable.

In the case of Relational Databases newer systems also provide horizontal scalability, Facebook is an example where Relational Databases are used in a horizontally scaled system consisting of many MySQL servers.

Among most important advantages of horizontally scalable systems is that they are very cost-efficient (Henderson, 2006).

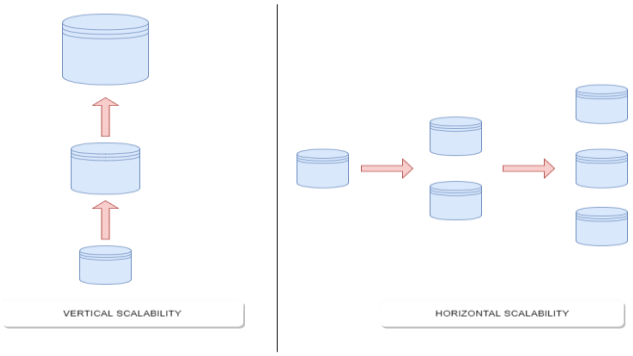

Figure 24, Vertical vs Horizontal Scalability

3. Elastic Scalability

Elastic Scalability is the ability of a system to be scaled both vertically and horizontally, or grow-shrink ability by adding or removing hardware nodes dynamically based on the needs of the application.

Cloud platforms mostly use elastic scalability, while most popular application of elastic scalability is Netflix, which uses Cassandra and HBase, which can be scaled dynamically without the need of re-sharding or rebooting.

Scalable Relational Database Management Systems

Generally, Relational Databases are considered as "one-size-fits-all" solution for data management, their maturity comes from decades of research and development.

Scalability has not been traditionally achieved with Relational Databases; however, with the increasing need for scalability, there have been some developments in this field such as MySQL Cluster, which is one of the first and most scalable relational systems even though it does not have high performance per node, compared to standard MySQL. Another system is Clusterix, which promises high scalability with a respectable per- node performance (Cattell, 2010) (Henderson, 2006).

In the following section, we take a look at MySQL Cluster, VoltDB and Clusterix as the three main and most important scalable relational solutions, while also making digressions and comparisons with NoSQL Systems such as Mongo DB and Cassandra.

MySQL Cluster

MySQL Cluster is part of MySQL release since 2004 and the code is based from an earlier project developed by Ericsson.

MySQL Cluster is available from MySQL and it is not open source, it works by sharding over multiple database servers while every shard is replicated so that recovery is supported and possible.

VoltDB

VoltDB oppositely to MySQL Cluster is an opensource solution, which is designed especially for high performance per node while also being highly scalable.

VoltDB's features include partitioning of tables over multiple servers while also allowing clients to call any server.

Distribution is transparent to the user, but the customer can choose the sharding attribute. In addition, replication of selected tables is possible while shards are always replicated so that data can be recovered if a node happens to crash.

Clusterix

Clusterix is very much like VoltDB and MySQL Cluster with the difference that Clusterix nodes are sold as rack-mounted appliance, they claim to be scalable to hundreds of nodes while having automatic sharding and replication.
Failovers and failed node recoveries are automatic, while a performance improvement is also reached by using solid-state disks (Cattell, 2010).
Same as other relational databases Clusterix is ACID compatible for transactions and supports SQL.

Experimental Work

This chapter is a benchmarking and performance testing of databases from two different types, the first database in study is MySQL as a representative of Relational Databases, while the second one is Mongo DB as a representative of NoSQL Databases.

The experiment is performed as follows:

1. Two scripts written in NodeJS perform all the CRUD operations, in two modes:
    a. Single Insert, Read, Update and Delete
    b. Concurrent Insert, Read Update and Delete
    c. Read, Update and Delete from multiple sources
2. All the data manipulated is generated by these scripts creating an id, company name and address.

The following hardware and software configuration is used:

Name	Configuration
CPU	Intel® Core™ i7- 75000U CPU @ 2.90 GHz
RAM	8GB
Hard disk	256 SSD
MySQL	5.7.21
Mongo DB	3.6
Node JS	8.11.1

Table 4, Hardware Configurations

Test Mode One: Single Insertion, Read, Update and Delete

The first part of the benchmark is testing of CRUD operation speeds while performing single operations.

As mentioned above a NodeJS script is used in each case (for both databases), which inserts a company id which is auto incremented, a given company name and an address. The script runs to a given amount of records to insert and is run using Windows command prompt.

In single insertion, all the records are inserted one by one and we do not have concurrency while the test is performed for two cases:

1. When the database is empty
    a. 1000, 10000, 100000 or 1000000 records are inserted one by one
2. When the database is already populated with a certain number of records

Reading is the second test performed, and in this case, we have the operation of reading all the records present in the database but accessing the database every time we want to perform a read.

1. Updating is also performed for one case:
    a. Update of all the records present in the database one by one.

Deletion is the last test performed for the first mode; in this case, the databases are tested in two cases:

1. Deletion of a single record with 1000, 10000, 100000 or 1000000 records in the databases
2. Deletion of all the records present in the database

Each test is performed three times and the average time is taken for the results.

Test Mode Two: Concurrent Insertion, Read, Update and Delete

The second part of testing consist of the same CRUD operations as in the first test however this time all the operations are performed concurrently.

Same as for the first test data insertion is performed through NodeJS scripts which generate the same data as before for two cases:

1. In an empty database
    a. 1000, 10000, 100000 and 1000000 records are inserted accessing the database only once for each number of records
2. When the database is already populated with data

Also, all the other tests are performed same as in the case of single insertions, while again each test is performed three times with the average time taken as result.

Test Mode Three: Read, Update and Delete from multiple tables/collections

The third and final test performed in this thesis is that of reading and then manipulating data from multiple tables in MySQL or collections in Mongo DB.

Multiple tables and collections are created manually, where data is also inserted individually in each of them, so for testing purposes only Read, Update and Delete are performed.

The following chapter presents the results of all these tests.

Benchmarking Results

Benchmarking results are represented in the form of tables and graphs representing each test.

The first part of the results is head-to-head comparison of single and concurrent insertion for both databases followed by the other operations in the following order: Read, Update and Delete.

Data Insertion

For testing data insertion, two tests are performed using two different NodeJS scripts. The first test is insertion of 1000, 10000, 100000 and 1000000 in an empty database where the database is accessed every time a new record is added, while the second test is concurrent insertion of data where the database is accessed only once at the beginning of the process.

The process of data insertion is completed with breaks after each test, meaning that after each pre-set number of records is inserted a complete wipe of the database is performed, so the database is empty at the beginning of each test.

Figure 22 shows the results of this test. As it can be seen, both databases perform quite well with Mongo DB being exponentially faster than MySQL.

Another important result that can be seen is the increase in time for both databases as the number of records is increased.

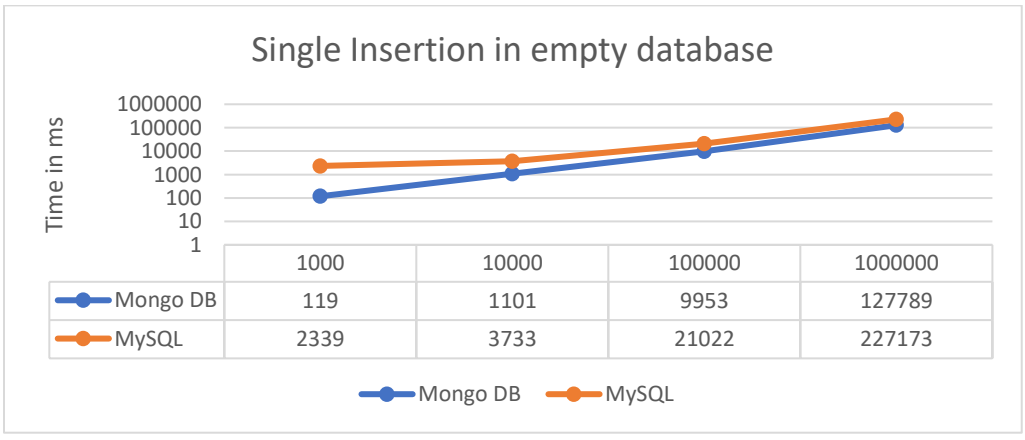

Figure 25, Single Insertion in an empty database Mongo DB vs MySQL

When trying to insert data concurrently the node script is such that all the requests are sent at the same time this is a very hardware- intensive approach and makes it very difficult for the CPU to handle everything at the same time.

The second test is concurrent insertion in an empty database. Figure 23 shows the results of this test where we can see that Mongo DB is faster than MySQL for each corresponding number of records inserted (1000, 10000, 100000 and 1000000).

Another important observation is that both databases fail when a larger number of records is inserted concurrently with Mongo DB failing to insert 1000000 records while MySQL fails at 57920 records every time is tested.

This failure in MySQL comes as a result of hardware limitations both in CPU speed and memory size, since when monitored with Windows Task Manager over 80% of RAM is used.

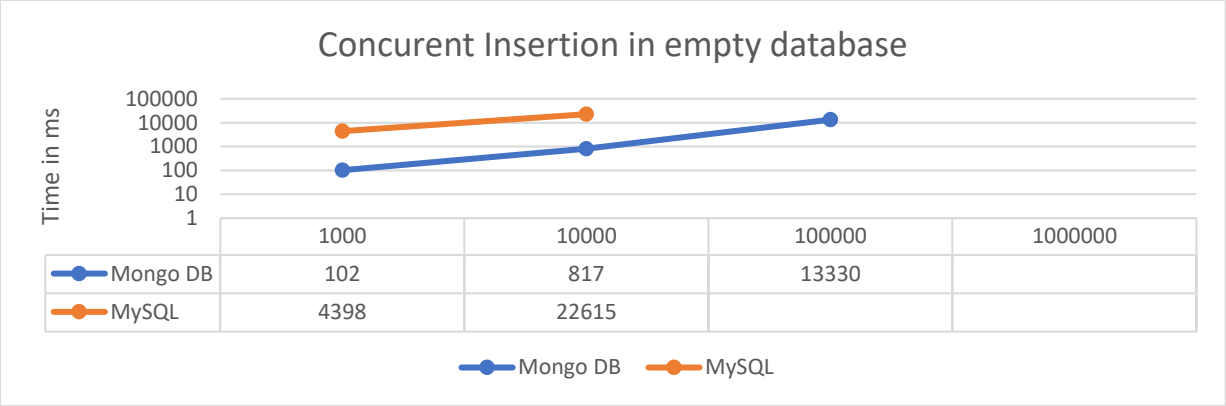

Figure 26, Concurrent Insertion in an empty database Mongo DB vs. MySQL

The third test performed for data insertion is single insertion in an already populated database.

Each number of records is inserted in the database as follows:

- 1000 records are inserted with: 1000, 10000, 100000 or 10000000 records in the database
- 10000 records are inserted with: 1000, 10000, 100000 or 10000000 records in the database

- 100000 records are inserted with: 1000, 10000, 100000 or 10000000 records in the database
- 1000000 records are inserted with 1000, 10000, 100000 or 10000000 records in the database

This test is performed for each database and the results are represented in the following figures (fig. 27 and fig 28).

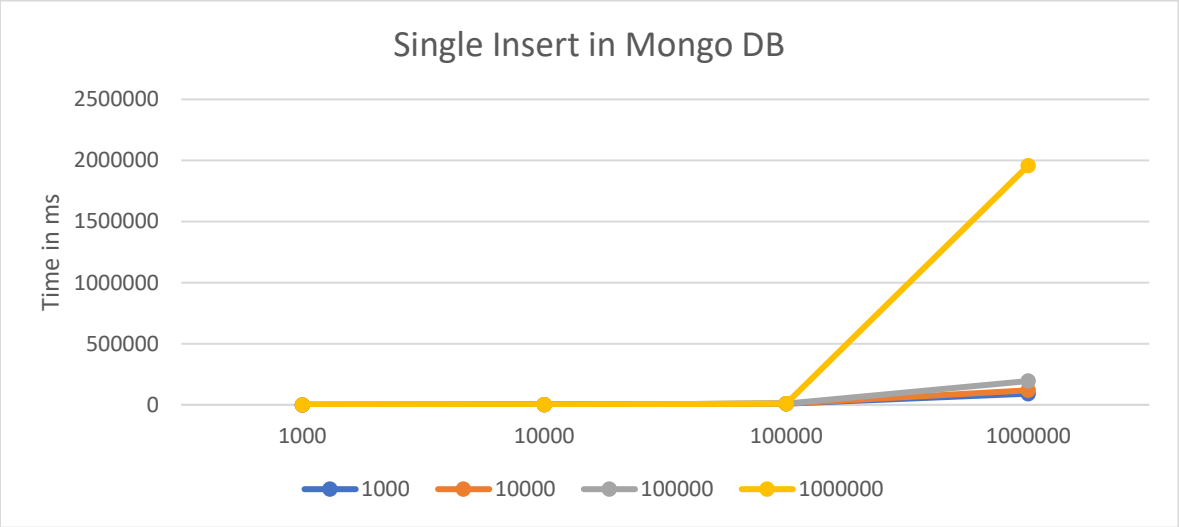

Figure 27, Single Insertion in Mongo DB in a populated database

From the results, it can be observed that Mongo DB is more stable and faster than MySQL.

Another important observation is that Mongo DB is not affected a lot by the number of records present in the database with the exception of the last case where the database is populated with 1 million records and another 1 million is inserted.

MySQL in the other hand is much more affected from the number of records in the database since execution time rises as the number of records present in the database rise.

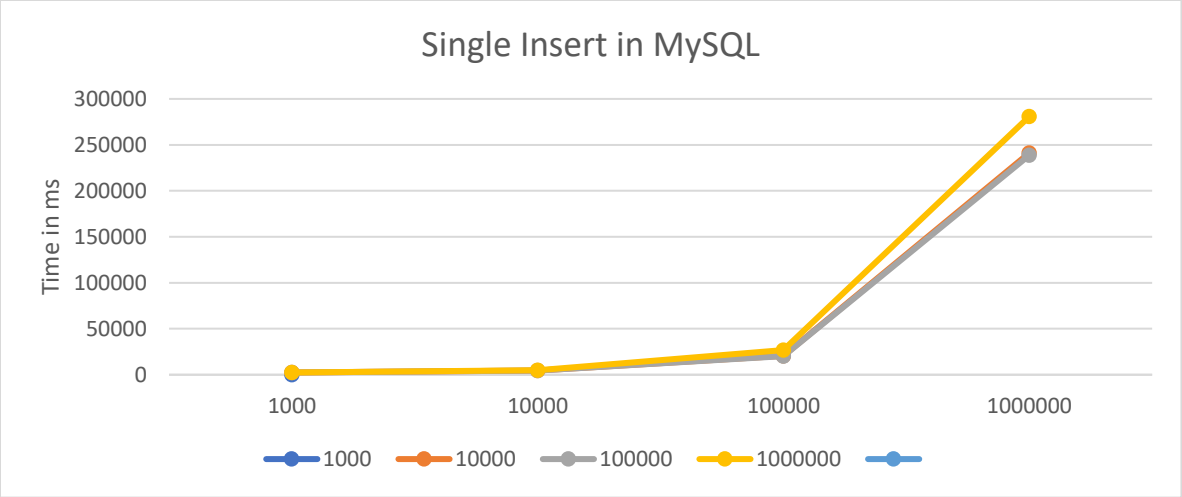

Figure 28, Single Insertion in MySQL in a populated database

All the results shown in the graphs above are also represented in tabular form below.

Type	Records to be inserted	Current Data	Mongo DB	MySQL
Single Insert in empty database	1000	0	119	2449
Single Insert in empty database	10000	0	1101	4733
Single Insert in empty database	100000	0	9953	21022
Single Insert in empty database	1000000	0	127789	227173
Concurrent Insert in empty database	1000	0	102	4398
Concurrent Insert in empty database	10000	0	817	22615
Concurrent Insert in empty database	100000	0	13330	✗
Concurrent Insert in empty database	1000000	0	✗	✗
Single Insert in populated database	1000	1000	128	2468
Single Insert in populated database	10000	1000	1075	11484
Single Insert in populated database	100000	1000	8896	20470

Single Insert in populated database	1000000	1000	91415	241185
Single Insert in populated database	1000	10000	134	2458
Single Insert in populated database	10000	10000	1133	4718
Single Insert in populated database	100000	10000	8993	20233
Single Insert in populated database	1000000	10000	118686	238757
Single Insert in populated database	1000	100000	140	2430
Single Insert in populated database	10000	100000	1180	4921
Single Insert in populated database	100000	100000	9290	26796
Single Insert in populated database	1000000	100000	194702	280808
Single Insert in populated database	1000	1000000	147	2442
Single Insert in populated database	10000	1000000	1186	4713
Single Insert in populated database	100000	1000000	10928	24380
Single Insert in populated database	1000000	1000000	1958214	278375

Table 5, Table of measurements for Insert operation

Update

Updating data or collections in a database is very sensitive since it requires completion of two actions instead of one, reading and writing.

In our tests, update is performed in two cases: Updating all the records in single mode and updating all the records concurrently.

The first graph below in figure 26 shows benchmarking results for single update where as it can be seen none of the databases fails however there is a difference in execution time where again

Mongo DB is faster compared to MySQL, with the biggest difference when 1 million records are updated.

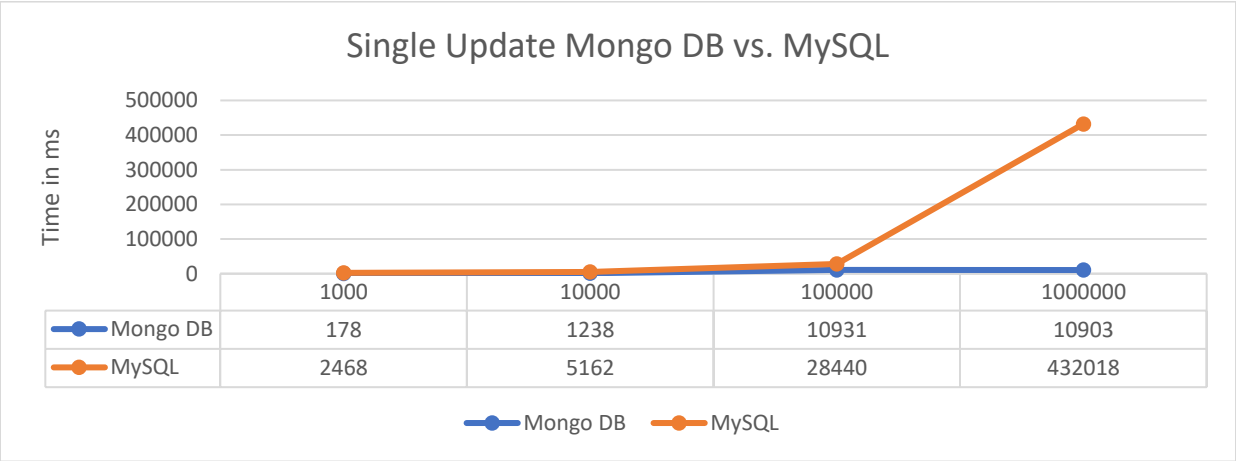

Figure 29, Single Update Mongo DB vs MySQL

The second test is concurrent update of all the data present in the database and the results are represented in the figure below.

As we can see both databases fail when updating 1 million records. As in the case of concurrent insertion, this comes mostly due to hardware limitations.

A speed difference is also noticeable with Mongo DB being much faster.

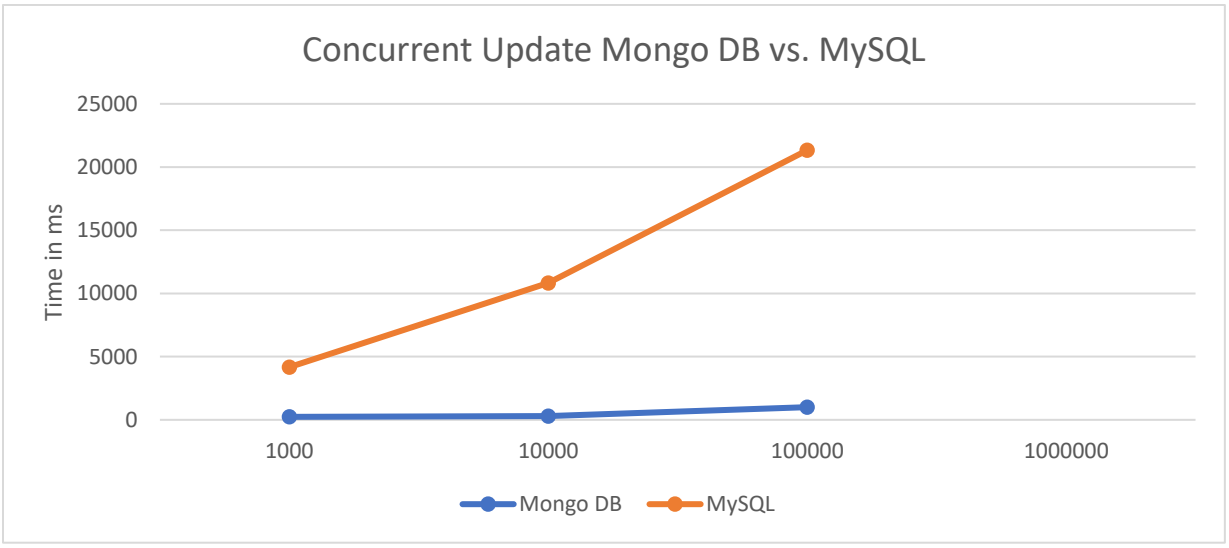

Figure 30, Concurrent Update Mongo DB vs. MySQL

Read

The third operation tested is Read. A similar approach is followed for Reading, as for Insert and Update with the difference that data is read concurrently from the database and not in single mode.

Another difference is an extra test performed. Reading speeds for a single record from the database when populated with 1000, 10000, 100000 and 1000000 records respectively.

Compared to other tests performed Read takes less time to execute. Following graphs show the results for both tests.

The graph in figure 28 shows the results for concurrent update for both Mongo DB and MySQL where it can be seen that Mongo DB performs faster than MySQL in most cases except for Reading of 1000 records.

Unlike other tests, this test was repeated more than 3 times with more or less similar results where every time MySQL is faster when reading a smaller number of records.

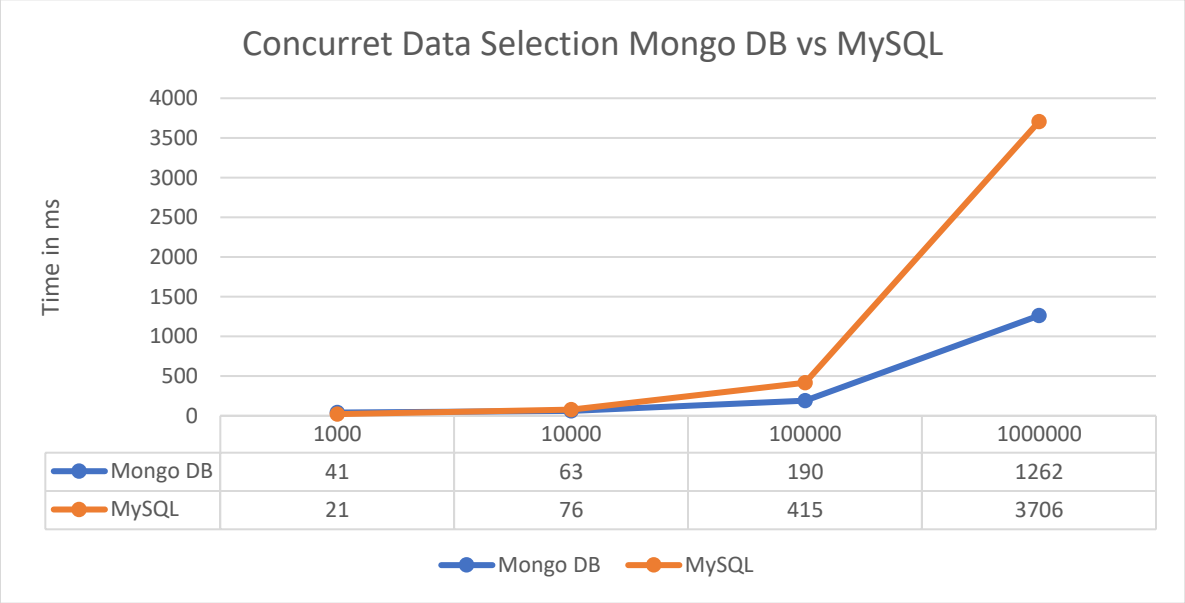

Figure 31, Concurrent Select Mongo DB vs MySQL

The results represented by figure 28 are the reason for performing the test of reading a single record from the database, results of which are represented in figure 29.

Results of this second test confirm the results of the previous one. MySQL is faster ever time read is performed for a single record.

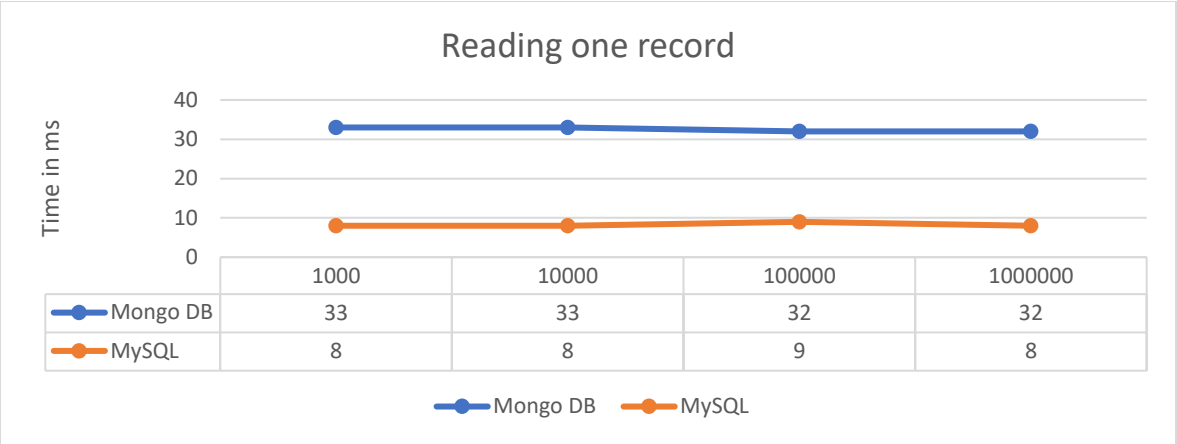

Figure 32, Single Record Reading

All the results and observations are presented in the following table.

Type	Objects to be read	Current Data	Mongo DB	MySQL
Concurrent Read	1000	1000	41	21
Concurrent Read	10000	10000	63	76
Concurrent Read	100000	100000	190	415
Concurrent Read	1000000	1000000	1262	3706
Reading one record	1	1000	33	8
Reading one record	1	10000	33	8
Reading one record	1	100000	32	9
Reading one record	1	1000000	32	8

Table 6, Table of results for Selection

Delete

Data deletion is the last test of this benchmark between Mongo DB and MySQL and the test is performed for two cases: Delete of all the records in the database concurrently and delete of a single record.

Results of these tests are represented in the figures below (Fig. 30, 31 and 32).

Figure 30 shows the deletion speeds between Mongo DB and MySQL when all the data in the database is deleted. Results show a difference in time between the two databases with Mongo DB being faster than MySQL.

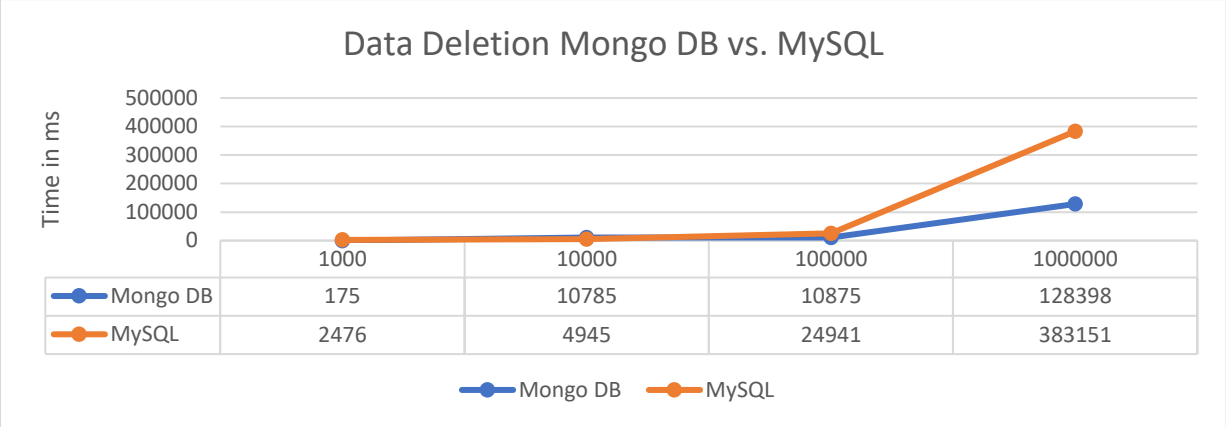

Figure 33, Data Deletion Mongo DB vs MySQL

As mentioned above, in the case of delete we have also tested the delete speed of a single record when we have a certain number of records in the database and the results are shown in the following graphs, where both databases perform very quickly and are stable throughout the test with the number of records present in the database almost not affecting the performance at all.

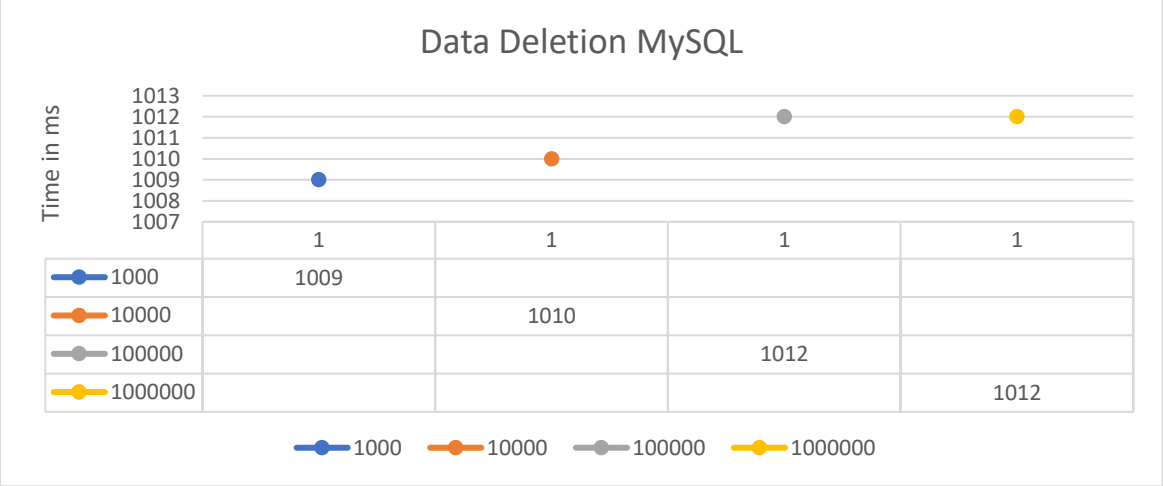

Figure 34, Data Deletion MySQL

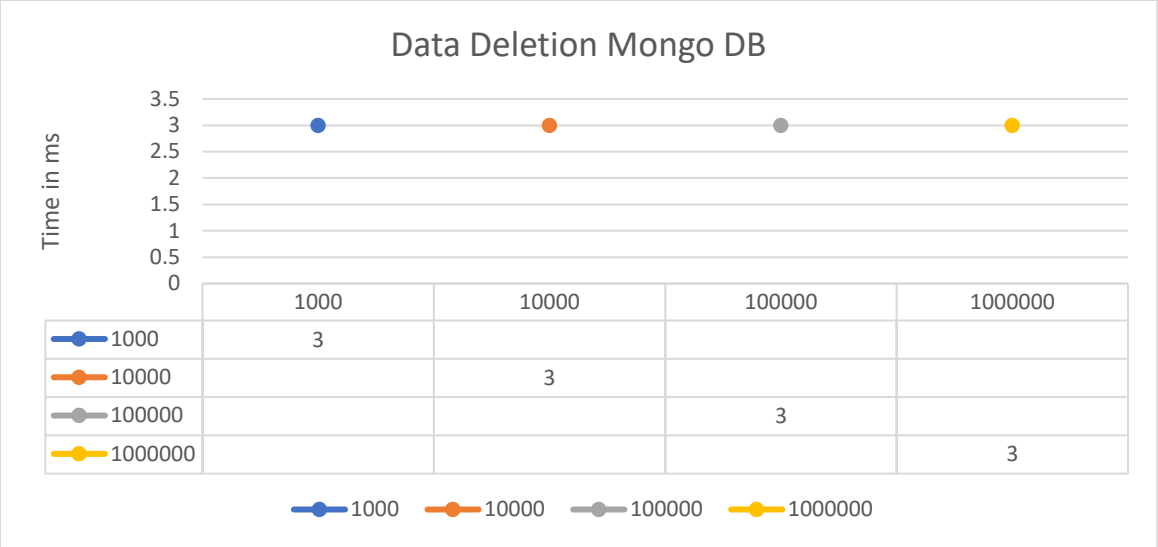

Figure 35, Data Deletion Mongo DB

All the results and observations are presented in the following table.

Type	Objects to be deleted	Current Data	Mongo DB	MySQL
Concurrent Deletion	1000	1000	175	2476
Concurrent Deletion	10000	10000	10785	4945
Concurrent Deletion	100000	100000	10875	24941
Concurrent Deletion	100000	1000000	128398	383151
Delete one record	1	1000	~ 3	1001
Delete one record	1	10000	~ 3	1001
Delete one record	1	100000	~ 3	1001
Delete one record	1	1000000	~ 3	1001

Table 7, Table of results for Delete Operation

Update from multiple tables/collections

The next test performed is more about testing on how databases perform when we deal with more complicated data models such as having to perform crud operations from multiple tables or collections.

Results from these tests are shown in the figures below where the first figure represents the data collected form updating data in multiple tables/ collections. When updating from multiple sources execution speeds are slower than when updating from a single table/collection.

Figure 33 shows that MySQL is slower than Mongo DB with the time difference increasing as number of records increases.

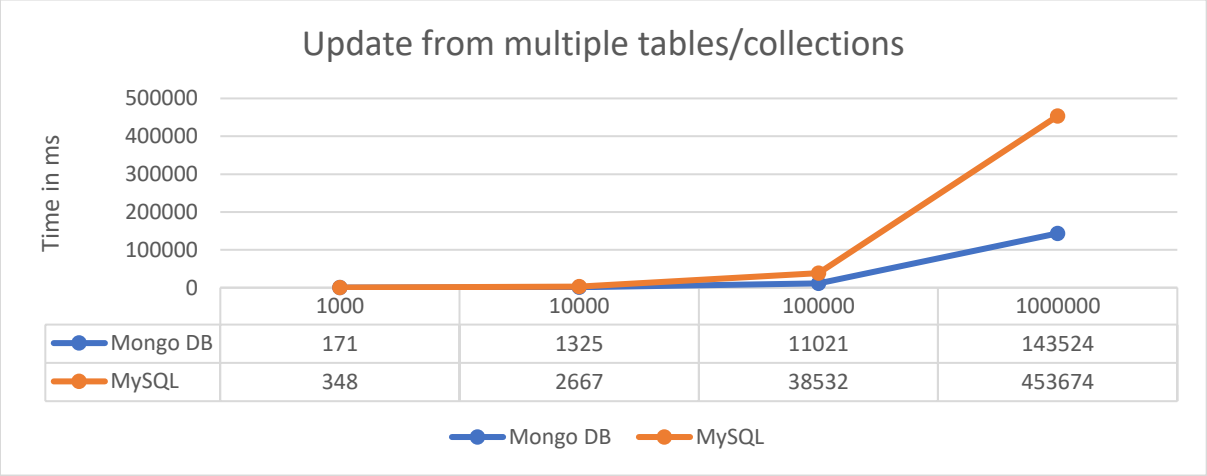

Figure 36, Update from multiple tables/collections

Read and Delete from multiple tables/collections

The same approach is used when testing Read and Delete operations from multiple tables/collections. The data is read from multiple sources and displayed as a single table.

Same as with update the read and delete operations are faster in Mongo DB with the only difference that Mongo DB slows down when having to delete large amounts of data.

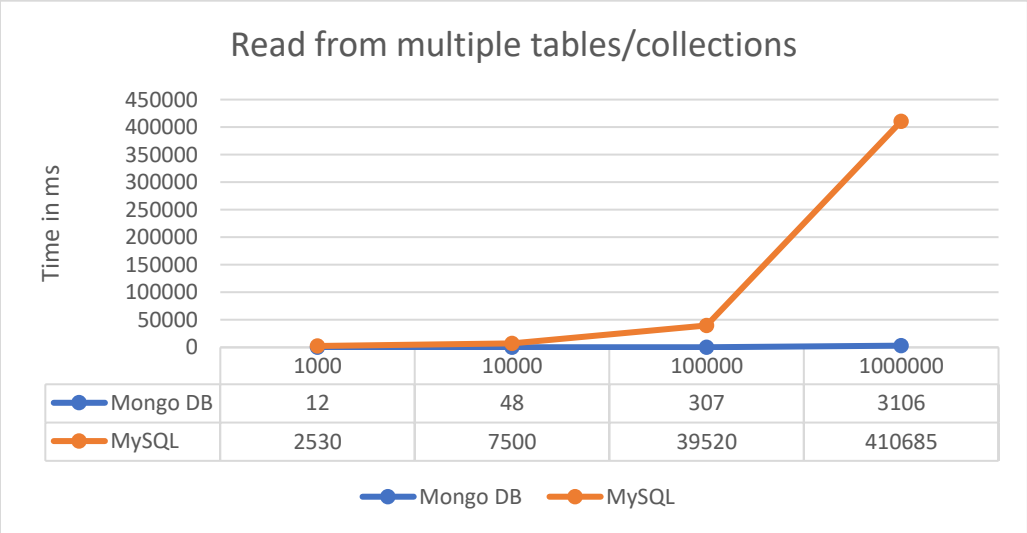

Figure 37, Read from multiple tables/collections

The test is performed multiple times with similar results; however, this delay might come because of hardware restrictions, while the read operation in Mongo DB is where the most difference can be seen since it is much more stable in all cases.

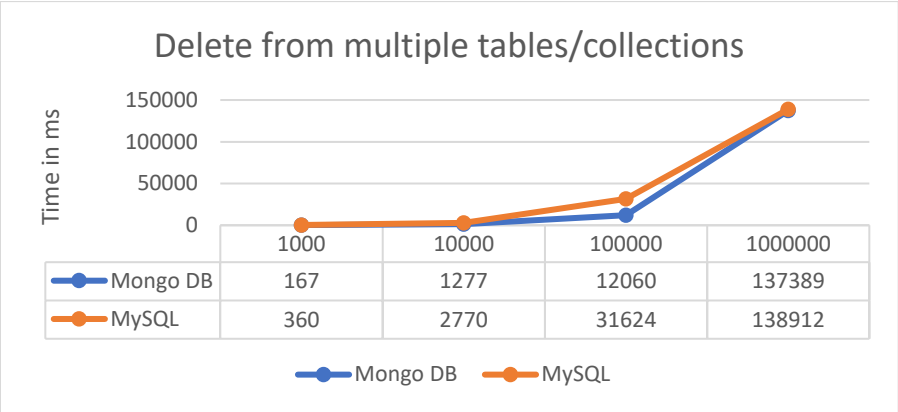

Figure 38, Delete from multiple tables/collections

Conclusions

The schema-less data model of NoSQL Databases ensures the ability to store any type of data without enforcing a strict data structure, while also ensuring that more sophisticated data models can be implemented. NoSQL Databases in general and Mongo DB in particular, are designed to prioritize availability over consistency, even though it varies a lot from the applications.

Relational Databases have a pre-set data model long before inserting data, which sometimes leads to complexity in development. A strong selling point for Relational Databases is the fact that they are mostly mature since a lot of development and research is conducted since they are an older technology compared to NoSQL Databases.

As mentioned in Chapter 3 of this thesis, Relational Databases are more about “What answers do I have?” compared to NoSQL ones which start at “What questions do I have?”

Apart from the theoretical part of this thesis, the last chapter is an experimental work in the form of a benchmark.

The benchmarking in this thesis has been done in a local environment, which is not the typical environment for NoSQL systems, however it provides a small-scale test for the theoretical claims and hypothesis of this thesis.

The results of these tests prove that Mongo DB performs better, since it has a better performance in terms of actions per time unit than MySQL.

The tests performed in this thesis are all done based on fairness and without any stereotypes while the results are those expected by the theoretical claims of this thesis as in many other related works.

In general, NoSQL databases are complementary to the relational model, and the work supports the main hypothesis that document–based databases are competitive to relational databases.

This thesis may also be expanded in the future since new scripts for more operations and database types can be easily added. In addition, the testing environment can be upgraded with the addition of more machines, which would provide a physical division rather than emulating it.

References

Abadi, D., 2007. *Column Stores For Wide and Sparse Data.* California, CIDR, pp. 292-297.

Beltrame, C., 2013. *Key-value stores.* Zurich, ETH- Zurich, pp. 1-12.

Bhardwaj, N., 2017. Comparative Study of Couchdb and MongoDB- NoSQL Document Oriented Databases. *International Journal of Computer Applications,* pp. 24-26.

Brewer, E., 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer Society,* pp. 23-29.

Bugiotti, F. & Cabibbo, L., 2013. A Comparison of Data Models and APIs of NoSQL Datastores.

Cattell, R., 2010. Scalable SQL and NoSQL Data Stores. *SIGMOD Record,* pp. 12-27.

Chandra, D. G., 2015. BASE analysis of NoSQL databases. *Elsevier- Future Generation Computer Systems,* Volume 52, pp. 13-21.

DB-Engines, 2017. *DB Engines.* [Online]
Available at: https://db-engines.com/en/ranking

Henricsson, R., 2011. *Document Oriented NoSQL Databases,* s.l.: Blekinge Institute of Technology.

Moniruzzaman, A. & Hossain, S. A., 2013. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application,* pp. 1-14.

Omji, M., Lodhi, P. & Mehta, S., 2018. Document Oriented NoSQL Databases: An Empirical Study. *Springer: International Conference on Recent Developments in Science, Engineering and Technology,* pp. 126-136.

Oracle, 2013. *Oracle: Big Data for the Enterprise,* s.l.: Oracle.

Pritchett, D., 2008. Base an ACID Alternative. *ACMQ,* pp. 50-55.

Redis, 2017. *Redis.* [Online]
Available at: https://www.redis.io

Schmitt, O. & Majchrzak, T. A., 2012. *Using Document- Based Databases for Medical Information Systems in Unreliable Environments.* Vancouver, s.n., pp. 1-10.

Seeger, M., 2009. Key- Value stores: a practical overview. *Medien Infromatik,* pp. 1-21.

Storey, V. & Song, I.-Y., 2017. Big data technologies and management: What conceptual modeling can do. *Elsevier: Data & Knowledge Engineering,* Volume 108, pp. 50-67.