Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

FACULTY OF MATHEMATICS & COMPUTER SCIENCE

NoSQL database systems for professional football analytics

Konstantinos Triantos (0852612)

March 8, 2017

2IMW05 - Capita Selecta Seminar Web Engineering
under supervision of assistant professor
dr. G.H.L. Fletcher

ABSTRACT

The evolution of needs demand the evolution of solutions. A new trend in database management suggests NoSQL databases as a solution in many data handling demands. SQL and NoSQL databases are different instances of the same solution. In this work, an attempt has been made to investigate, which flavor of NoSQL database is more appropriate for each situation and which are their similarities with the SQL databases. More specifically, the intention of this work is to define which type of store is the most appropriate for a specific input data-set related to professional football analytics based on the paper "Soccer Video and Player Position Data-set". According to this specific data-set, the most popular NoSQL stores are examined, in terms of scalability, query language, users community and availability. The sample which is investigated consists of document stores (MongoDB, Azure DocumentDB, Apache Cassandra), stream stores (Apache Storm and Apache Spark) and graph stores (Neo4J and OrientDB). Regarding data-stores, which use documents, MongoDB is the most popular store. After investigating, stream stores, which seem more real-time business systems than to regular database systems, no conclusion can be made since Storm and Spark are similar. About graph data-stores, which use graph structures for semantic queries, Neo4J is more popular than OrientDB, but OrientDB has a slight advantage against Neo4J in terms of scalability. Because of the fact that in many cases the traditional relational databases still seem to be the ideal solution an experiment has been taken place, in order to specify which solution fits better to the examined data-set. According to these aspects, a set of benchmark queries for MongoDB and MySQL, has been built. This specific benchmark covers a set of requirements regarding querying the input data-set and the output of the experiment is that both SQL and NoSQL have advantages and disadvantages against each other. More specifically, according some benchmark queries NoSQL behaves better than SQL and according some others vice versa.

CONTENTS

1 Introduction

1.1 The need of Database Management

Since the begging of computer age, the most crucial and essential component regarding computer's performance has been the memory. More specifically, the manner in which the information is organized inside the memory, can conclude either the success either the disaster of any computation attempt. Nowadays, data management is a hot topic, since almost any application is relied to data, which size increases exponentially according the time. From simple web sites to complex business analyst tools, different types of data should be processed, stored and retrieved, in a efficient and accurate way. As a result, the systems, which operate over data must be adaptive, responsive, efficient and accurate as well.

1.2 What is Database Management System

Any system, which operates on data management is called Database Management System (DBMS). This kind of systems is characterized as a high-level software, which cooperates with low-level interfaces, aiming on data storage and querying. More specifically, a DBMS assists the user to handle enormous collections of data, which are stored in the hard drive. Because of the fact that a data collection does not have standard shape or size, a lot of DBMS have been developed, during the years in order to assist in dealing with this variety in data format. Consequently, there are dozens of solutions, which have been developed, according the time and offer different approaches in data management, according to user's needs. However, only a relatively small set became popular and stay in use for a longer time. Precisely, the most popular DBMS all this time is the Structured Query Language (SQL) database systems.

1.3 What is a SQL Database System

Structured Query Language (SQL) databases are the primary data storage mechanism in recent times. An SQL database is based on the relational algebra schema, which organizes the data into tables with specific structures and attributes. Nowadays, there are a lot of versions of SQL database systems such as MySQL, PostgreSQL and SQLite. These specific versions offer various functions and tools regarding data management and consists of the biggest piece of the pie chart of popular database systems.

However, the new trends in web applications and the increase of data size, unravel new needs, which demand new approaches in data management. On the other hand, the idea of SQL appeared four decades before, when the data needs and the application were different. As a result, in order to solve these different kind of problems, for decades SQL databases evolve and new software have been developed along with the new trends in computer applications. Unfortunately, there are still needs that an SQL database cannot treat efficiently and accurate.

1.4 WHAT IS A NOSQL DATABASE SYSTEM

The evolution of needs demand the evolution of solutions. A new trend in database management suggests NoSQL databases as a solution in many data handling demands. NoSQL gain traction day by day, offering an alternative way on data management with popular options such as MongoDB, CouchDB and Apache Cassandra.

The main characteristic of the NoSQL databases is the fact that they operate to data without using the traditional relational algebra schema. Thus, a NoSQL user has the ability to interact with data without having to define or deal with a predefined structured schema. In addition, NoSQL databases run on clusters, which means that they distribute data to multiple database instances with no shared resources. As a result, in contrast with SQL databases, the stored data can be reproduced and copied to more instances and be supported by more than one servers. NoSQL databases interesting for investigation for two major reasons:

Productivity During application's development, much effort is put, on mapping the input data in structured schema. A lot of assumptions take place and a lot diagrams are created in order to produce the best relational schema for the data. By using schema-less NoSQL database management systems, the database system adapts its schema to the data and not the data to the schema. More specifically, NoSQL database offers a data model that fits to the application's need. As a result, a programmer spends his effort on other aspects instead of trying to simplify the data, in order to fit in a standard schema.

Scalabilty According the time, the size of data is increased in terms of volume. As a result, the requirement of fast and efficient querying becomes more expensive. The main reason is the fact that SQL databases run on a single server. On the other hand, NoSQL databases run on clusters and thus they can split the load into two or more servers. As a result, the demand of large-scale data handling can remain cheap.

1.5 SQL VS NOSQL

SQL and NoSQL databases are different instances of the same solution. In data world, there problems that SQL databases seems to be ideal and problems that NoSQL databases treat better. The main difference between SQL and NoSQL databases is the fact that NoSQL databases offer a variety of stores. Each store is different than the other and is built to treat a specific problem. Thus, a NoSQL database can be introduced as a specialist for a specific problem.

Nowadays, in data science field, there are a lot and different challenges and even more situations of problems. SQL databases consists of a general solution, which does not adapt to the problem that solves. On the other hand, NoSQL databases consists of a "tool case", which offers the proper tool according the problem.

In this work, an attempt has been made to investigate, which flavor of NoSQL database is more appropriate for each situation and which are their similarities with the SQL databases. In a later follow-up chapter, an input data scenario is introduced to which we determine the optimal choice. More specifically, in the following chapters, there will be an introduction of the most relevant NoSQL database management systems and the logic behind how these databases organize the data.

2 Concept Idea

The intention of this work is to define which type of store is the most appropriate for a specific input. More specifically, this specific input is a data-set related to professional football analytics and is described in details in Chapter 3. According to this data-set,the most popular NoSQL stores will be examined, in terms of scalability and productivity.

2.1 Scalability

In recent times, many sub-fields of scientific community define scalability as the capability of a system to amplify in order to fulfill new requirements and demands. For instance, scaling a web-based application is the process regarding allowing more people to use this web-based application. However, the term scalability is inherently a bit amorphous and typically dependent on a specific use case. Within this discussion, scalability is defined as the ability to add computational resources to a database in order to gain more throughput. Regarding this definition, there are two major types of scalability available - vertical and horizontal.

Vertical scalability indicates the increase of the resources within the same logical unit to increase the capacity. For instance a vertical scalability action involves moving from one machine to another that has more capacity (RAM, CPU, storage or a combination of these resources). Thus, it is obvious that this approach is more naive and scaling a database vertically is expensive in both financial outlay and resource dedication. In considering financial outlay, larger, more robust hardware is expensive to acquire and operate. In terms of resource dedication, if there are any requirements for maintaining up-time, significant operational planning and effort are usually required to migrate to the new system. If the volume of data is large, then the physical transfer from the old system to the new can take an inordinate amount of time depending on the load.

In contrast, a system is horizontal scalable if hardware can be added incrementally. More specifically, for more capacity, additional hardware should be added. In an ideal horizontally scalable system, addition of hardware should provide linear increases in capacity available without reconfiguration or downtime required of existing nodes [1]. The difference between these two approaches can be visualized by the Figure 2.1.

2.2 Productivity

As productivity, it is defined how handy and useful the database software is. More specifically, with the term productivity, the tools that a database system offers to its user to assist him, are indicated. Productivity consists of a major metric of evaluation, because a more friendly-user system is more preferable than another one, even with more utilities and better performance [2]. Examples of productivity of a database system is the practicality of this database's query language and the database's availability in terms of supported libraries and forums.

Regarding the query language, an important notion should be mentioned. Considering the topic of this study, as in database systems, there is no ideal query language, which will fit the needs for all possible developers under all possible conditions. Because of the fact that humans and problem domains vary, it seems impossible to create a metric, which will

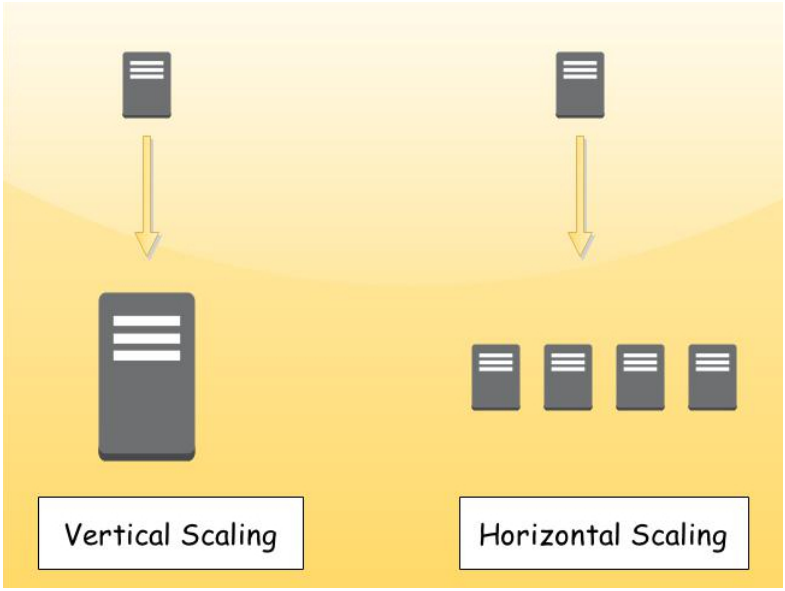


Figure 2.1: Vertical vs Horizontal Scalabilty [3]

indicate if a query language is better than another one. As a result, the actual metric, which will be used on query language evaluation, is the similarity of this specific language with the SQL query language, since SQL is standard for decades.

2.3 METRICS

As it is mentioned in the previous sections, each store will be examined according its scalability and its productivity. More specifically, the examined stores are investigated, in terms of:

- ***Scalability***: In scalability sections an attempt is made to investigate how each store can scale, with priority to horizontal scale.
- ***Query language***: Each query language section consists of an introduction to query language of a specific store. The basic queries are introduced in a form of tutorial and when the query language is not similar to SQL, an attempt is made to parallel the basic commands with the SQL ones.
- ***Users community***: Users community section introduces the available communities for each data store. More specifically, the research focuses on the popularity and the size of each community.
- ***Availability***: This section is presented with the previous one, as one sub-chapter and it focuses on third party libraries and drivers that support programming languages.

3 Examined Data-set

3.1 Soccer Video and Player Position Data-set

In the paper "Soccer Video and Player Position Data-set"[4], an attempt is made to present a data-set of body-sensor traces and corresponding videos from several professional football matches. More specifically, this data-set has been obtained from football matches, which took place in the home ground of the Norwegian professional team Tromso IL, during November 2013. Precisely, during the matches, all the players of Tromso IL were recording their positions with a use of a GPS device. The input data-set is obtained by the Europa League Group stage match (Group K) on Thursday 7^{th} November 2013 at 21:05CET in Alfheim, Tromso.

In order to elaborate the process; each player was wearing a belt during the match and a tracking system was providing his speed, his acceleration his force and his coordinates on the field, followed by his ID and a time-stamp [4]. Each player's position was measured at 20 Hz using the ZXY Sport Tracking system.

More specifically, ZXY tracking system provides a two-dimensional positional coordinate system; which is calibrated to a station. At Alfheim stadium (home ground of Tromso IL), the coordinate system looks like the positive part of a Cartesian space. The positive x-axis points southwards parallel along the long side of the field, while the positive y-axis points eastwards parallel with the short edge of the field, as shown in Figure 3.1.

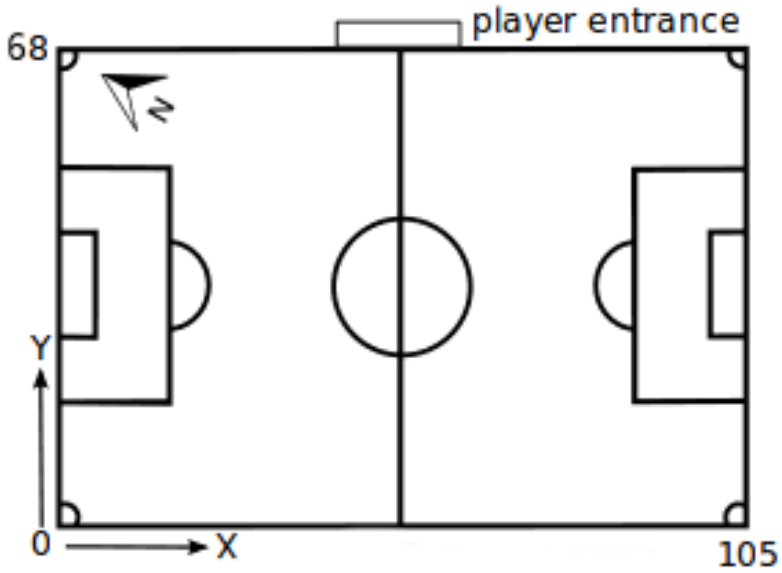


Figure 3.1: Layout at the Alfheim Stadium [4]

As we can see, the position $(0,0)$ is located in the north-western corner of the football field, which from the position of the cameras is the lower-left corner. The football pitch is 105x68 meters wide and as a result the valid in-field values for the data set fields x_pos and y_pos are in the values range of $0 \leq$ x_pos ≤ 105 and $0 \leq$ y_pos ≤ 68.

3.2 DATA-SET FORMAT

The logs extracted from the ZXY Sport Tracking system are stored in files. Each line of a file contains one time-stamped record from exactly one belt. Belts are uniquely identified by a tag-id, and each player had been wearing only one belt. The tag-ids had been randomized for each game to provide a higher-level of anonymity to the players. The data records The are saved in a file, which is CSV (comma-separated values) format and have the following format:

time (string) - Local Central European Time (CET) time encoded as ISO-8601 format.

tag_id (int) - The sensor identifier.

x_pos (float) - Relative position in meters of the player in the field's x-direction.

y_pos (float) - Relative position in meters of the player in the field's y-direction.

heading (float) - Direction of the player is facing in radians.

direction (float) - Direction of the player is traveling in radians.

energy (float) - Estimated energy consumption since last sample.

speed (float) - Player speed in meters per second.

total (float) - The number of meters traveled so far during the game.

A sample of the recorded values is presented below by the Figure 3.2.

```
'timestamp','tag_id','x_pos','y_pos','heading','direction','energy','speed','total_distance'
    ...
'2013-11-03 18:30:00.000612',31278,34.2361,49.366,2.2578,1.94857,3672.22,1.60798,3719.61
'2013-11-03 18:30:00.004524',31890,45.386,49.8209,0.980335,1.26641,5614.29,2.80983,4190.53
'2013-11-03 18:30:00.013407',0918,74.5904,71.048,-0.961152,0,2.37406,0,0.285215
'2013-11-03 18:30:00.015759',109,60.2843,57.3384,2.19912,1.22228,4584.61,8.14452,4565.93
'2013-11-03 18:30:00.023466',909,45.0113,54.7307,2.23514,2.27993,4170.35,1.76589,4070.6
    ...
```

Figure 3.2: Samples obtained by the 20 Hz GPS sensor traces [4]

The logs from the ZXY Sport Tracking system, which will be used in this research have the form which is presented in table 3.1. More specifically, this is the data-set which will be used in any experiment during this work.

Table 3.1: Input data-set format

time	*tag_id*	*x_pos*	*y_pos*	*heading*	*direction*	*energy*	*speed*	*total*
2013-11-[..]	6	50.173	7.49273	-2.74995	2.78075	1098.31	1.92885	1073
2013-11-[..]	1	62.769	13.7037	-1.91762	1.68409	1151.91	0.94616	871.1
2013-11-[..]	4	66.44	34.7657	-2.02796	2.28517	1215.87	0.8955	832.4

...

4 Document Stores

The first examined kind of storage refers to document stores. The structural node of a document store is the *document*. In a document database, documents are the smallest units, consisting of a *collection*, which build the entire database, in the same trend. The structure of this schema can be seen in Figure 4.1:

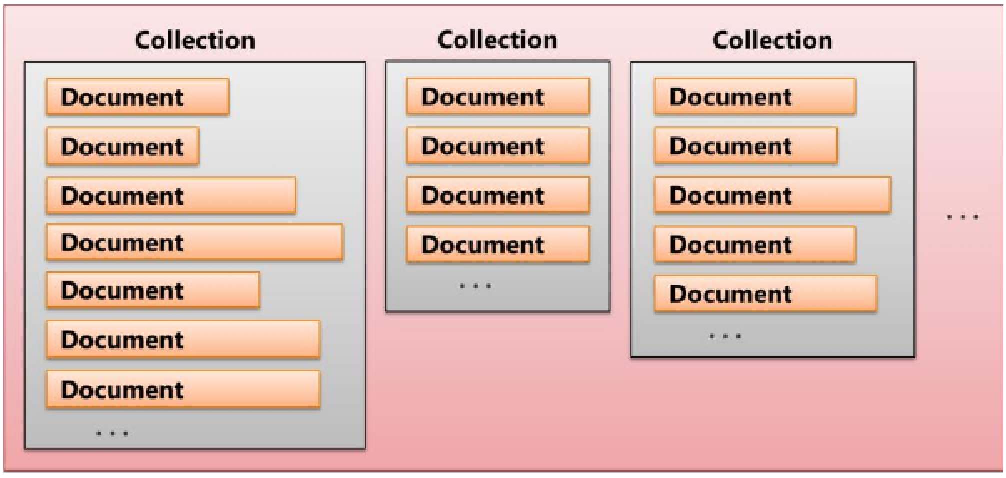
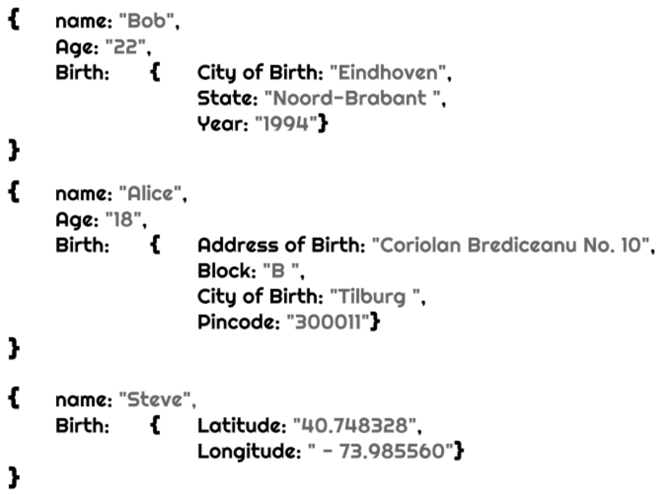


Figure 4.1: A document database contains collections of JSON documents [6]

If we would like to make a parallelism, a document is equivalent to a row in a SQL table. and, in the same fashion, a SQL table is equivalent to a collection in a document database. However, there is one significant difference. In contrast with a record in a SQL table, a document of a collection does not need to be in the same form as the rest of the documents. More specifically, each of them can have different structure with different fields [5]. For instance, let's consider the collection of TU Eindhoven students, which is presented by the Figure 4.



In this example, it is obvious the fact that while the three documents all represent people and their birth locations, the representative models are different. A significant difference between a key-value store and a document store is the fact that a document stores embeds attribute metadata associated with stored content. This architecture provides a way to query data based on the contents. As a result, in the above example, a user could search for all documents in which attribute "City of Birth" is "Eindhoven" that would deliver a result set containing all documents associated with any "name" that is in this particular city [6].

4.1 SUITABLE CASES

Document databases are suitable for 4 kind of cases [7]:

Event logging Many kinds of applications have logging needs. However, not all of them have the same ones. A document database can store all these different types of events because of its schema and become a central store for them.

Content management systems and Blogging platforms Because of their schema document databases can understand JSON documents, which are the most common in content management systems and applications such as web-sites for blogging.

Web analytics and Real-time analytics Since there is no need for schema changes when there are new incoming data, document databases are suitable for tracking metrics in real-time conditions.

E-commerce applications E-commerce use to need flexible databases which can evolve with not expensive costs.

4.2 UNWARRANTED CASES

On the other hand there are cases, when a document database is not the most efficient solution [7].

Event logging Many kinds of applications have logging needs. However, not all of them have the same ones. A document database can store all these different types of events because of its schema and become a central store for them.

Content management systems and Blogging platforms Because of their schema document databases can understand JSON documents, which are the most common in content management systems and applications such as web-sites for blogging.

EXAMINED DATABASES SYSTEMS

In the following sub-chapters, three document stores are introduced:

- **MongoDB**
- **Azure DocumentDB**
- **Apache Cassandra**

4.3 MONGODB

As the official website claims "MongoDB is a document database that provides high performance, high availability, and easy scalability" [8]. Implemented in C++, MongoDB is a product of MongoDB Inc, which released first time in 2009 [9]. Its data schema has the same structure with the general document store. A MongoDB database includes a set of collections. Each collection includes a set of documents. Every document is a set of key-value pairs and has the introduced above dynamic schema. Dynamic schema means that a collection of documents is able to hold different types of data. The investigated MongoDB release is the 2.4, 2014.

4.3.1 QUERY LANGUAGE

MongoDB queries are represented with a JSON structure. In order to build a query, a document with the preferred to be shown properties needs to be built up. This specific document will be compared with the database records and the results that match will be projected. Something that must be mentioned is the fact that MongoDB treats each property as having an implicit boolean AND and, in addition, it supports also boolean OR queries. In addition to exact matches, MongoDB has operators for greater than and less than, regarding values with order. [10] A sample query document, which is useful to understand the above description, is the following:

$$q = \{"firstname" : "Bob"\}$$

The above query q matches all documents in a collection with firstname equal to "Bob". If there is need to retrieve all documents with firstname "Bob" AND city of birth "Eindhoven", the query q would have the following form:

$$\begin{aligned} q = \{&"firstname" : "Bob", \\ &"state" : "Noord-Brabant"\} \end{aligned}$$

Something that must be mentioned is the fact that the comma inside the query q represents the logic AND. Taking into account the situation that we wanted to retrieve all documents with a age value of greater than 18, we would write:

$$q = \{"age" : \{"\$gt" : 18\}\}$$

All the above queries can be implemented with the use of $db.collection.find()$ method. This specific method retrieves documents from a collection and returns a cursor to the retrieved documents. For instance, if we would like to retrieve from a collection with named "students", all the students with their year field with value less than 1994, the syntax would be the following:

$$db.students.find(\{year : \{\$lt : 1994\}\})$$

4.3.2 SQL TO MONGODB MAPPING

The biggest portion of data science community started learning the database technology with the use of relational databases such as SQL. Because of this fact, in this chapter an attempt is made to map SQL's query language to MongoDB's query language, in terms of basic commands. The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts:

Table 4.1: SQL terminology & concepts to the corresponding MongoDB ones.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document
column	field
index	index
table joins	embedded documents and linking
primary key	primary key
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators:

Table 4.2: SQL aggregation terms, functions & concepts into MongoDB ones.

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum
JOIN	No direct corresponding operator; however, the $unwind allows for somewhat similar functionality.

4.3.3 SCALABILITY

Taking into consideration the year 2016, thousands companies and organizations use MongoDB in order to construct top-performance systems at scale. More specifically, the number of web companies that rely their projects on MongoDB, increases according the time. These project have a variety of needs: from single server to cloud clusters with over a thousand nodes, executing millions of operations per second over petabytes of data [11]. The official web-page of MongoDB claims that MongoDB fulfills the following aspects in terms of scalability:

- **Cluster Scale**. Companies such as EA Sports FIFA, eBay and Sporting Solutions "distribute their database across 100+ nodes, often in multiple data centers" [11].
- **Performance Scale**. Foursquare, AHL and many more companies "Sustain over 100,000 database read and writes per second while maintaining strict latency SLAs" [11].
- **Data Scale**. McAfee Global Threat Intelligence, Adobe and CARFAX "store over one billion documents in their database" [11].

Scaling of MongoDB can be done via Ops Manager, included with MongoDB Enterprise Advanced. This software offers many powers like the ability to deploy and scale shared clusters across multiple data centers. In addition with this Manager, MongoDB directors try to support users officially with guides and tutorials regarding scalability at offering the following link:
https://www.mongodb.com/collateral/mongodb-performance-best-practices

4.3.4 AVAILABILITY & COMMUNITY

MongoDB and mongodb.org supported drivers is an open source software, which is hosted at Github. This is the reason that its community is large in terms of active members and activities [13] [14] [15] [16] [17]. In addition, MongoDB community encourages the user to join them using the following web-page: https://www.mongodb.org/get-involved.

Regarding third party libraries, MongoDB supports the following programming languages offering proprietary protocol using JSON.

- Actionscript
- C
- C#
- C++
- Clojure
- ColdFusion
- D
- Dart
- Delphi
- Erlang
- Go
- Groovy
- Haskell
- Java
- JavaScript
- Lisp
- Lua
- MatLab
- Perl
- PHP
- PowerShell
- Prolog
- Python
- R
- Ruby
- Scala
- Smalltalk

4.4 AZURE DOCUMENTDB

Azure DocumentDB is a NoSQL document database designed, by Microsoft, and released on 2014. DocumentDB is a service, which is built up from the ground up in order to support JSON and JavaScript. DocumentDB's data model is the same as the general schema which is introduced in the beginning of this Chapter [chapter 4]. As a result, every record inside this database is stored as a JSON document. The official documentation describes it as the "right solution for applications that run in the cloud when predictable throughput, low latency, and flexible query are key". A well-known user of DocumentDB is the Microsoft MSN, which supports daily millions of users [18].

4.4.1 QUERY LANGUAGE

Users of DocumentDB service use an extended subset of SQL, in order to query information from the database. To clarify this, the following example is considered:

Consider a JSON document, which refers to a person. This simple JSON document is about a student of a university, with name "Bob". The document has strings, numbers, boolean, arrays and nested properties. All these aspects can be seen in Figure 4.2:

Figure 4.2: The JSON document that represents the student "Bob".

In order to retrieve the university in which student Bob studies, the query that will be built, will be following the regular SQL syntax rules. More specifically, this specific

SELECT s.university
FROM students s
WHERE s.name = "Bob"

As a database user who knows SQL can probably Figure out that

- SELECT requests the value of the element *university*
- FROM indicates that the query should be executed against documents in the students collection

- WHERE specifies the condition that documents within that collection should meet

The query's result is *university* attribute for Bob formatted as JSON data:

```
{
 "university" : Eindhoven university of Technology
}
```

4.4.2 SCALABILITY

A single DocumentDB database can scale practically to an unlimited amount of document storage partitioned by collections. As a result, there is no special treatment regarding scalability and the user can scale just by adding more collections.

Each collection provides 10 Gigabytes of storage, and an variable amount of throughput. A collection also provides the scope for document storage and query execution; and is also the transaction domain for all the documents contained within it [20].

4.4.3 AVAILABILITY & COMMUNITY

Azure DocumentDB is a service for commercial use. As a result, its community is smaller than MongoDB or Cassandra. However, Microsoft provides professional support to each user. DocumentDB can be used within Windows and Linux environments and up to now the supported programming languages are:

- .Net
- C#
- Java
- JavaScript
- Python

Whatever choice the developer makes, the client accesses DocumentDB through RESTful access methods. A developer can use these to work with documents in a collection in a few different ways [19]. These options are presented bellow:

- Using these access methods directly for create/read/update/delete (CRUD) operations.
- Submitting requests expressed in DocumentDB SQL.
- Defining and executing logic that runs inside DocumentDB, including stored procedures, triggers, and user-defined functions (UDFs)

4.5 APACHE CASSANDRA

Apache Cassandra is an open source distributed database management system designed to handle large amounts of data across many servers. Cassandra is document store that offers robust support for clusters spanning multiple data-centers, with asynchronous master-less replication allowing low latency operations for all clients and with a high value on performance. Consisting of a massively scalable NoSQL database, Cassandra can be found at companies that focus on big data, such us Amazon, Google and Facebook.

The architecture of Cassandra makes this NoSQL database to be able to scale, perform, and offer continuous availability because of the fact that it has been built from the ground up with the understanding that hardware and system failures can and do occur. This assumption concludes that Cassandra uses a different way of managing and protecting data than the traditional relational database management systems.

Cassandra has a peer-to-peer distributed architecture which is easy to set up and maintain. In a Cassandra database system, all nodes are the same in a sense that there is no concept of a master node and all nodes communicate with each other via a protocol. With the use of this architecture Casandra is capable of handling petabytes of information and thousands of concurrent users/operations per second as easily as it can manage much smaller amounts of data and user traffic. Moreover, unlike to other systems, Cassandra has no single point of failure and therefore is capable of offering real and continuous availability [21].

4.5.1 QUERY LANGUAGE

In comparison with MongoDB and DocumentDB, the main difference is that Cassandra does not support joins or subqueries, except for batch analysis through Hive. Instead, Cassandra emphasizes denormalization through CQL features like collections and clustering specified at the schema level.

CQL is the recommended way to interact with Cassandra. CQL is a mechanism ,which consists of statements and is similar to SQL. As in SQL, some statements directly change data, other look up data, and some other change the way data is stored. The simplicity of reading and using CQL is an advantage over older Cassandra APIs. The main goal of this approach is to give the end user a familiar feel of working with SQL. Additionally CQL is useful in knowing the information related to cluster, Keyspaces structure. And it also provides commands to read the TTL(Time To Live) for data and many other much useful information

In order to understand how CQL works, let's consider the following example. From a database there is a need to retrieve all students with the name Bob and age 22 years old. In order to retrieve these results, we have to use the SELECT command.

SELECT *
FROM users
WHERE name = 'Bob' AND age = 22;

Similar to a SQL query, the user can use the WHERE clause and then the ORDER BY clause to retrieve and sort results.

4.5.2 SCALABILITY

The official web-page introduces Apache Cassandra database as the "right choice when you need scalability and high availability without compromising performance" [22]. More specifically, directors advocate that Cassandra's linear scalability and its fault-tolerance on commodity hardware or cloud infrastructure indicate it as the best platform to process mission-critical data [22].

Apache Cassandra fulfills the obligations of an ideal horizontally scalable system, in a sense that allows seamless addition of nodes. As a result, if there is a need for more capacity, extra nodes can be added to the cluster and then this cluster will utilize the new resources automatically [1].

An example which proves that Cassandra adapts easily and effective on scale requirements is Netflix. Netflx has been rolling out the Apache Cassandra NoSQL data store for production use. Benchmark tests validated that Cassandra tends to outpace its NoSQL competitors in performance for many use cases [23]. As it mentioned in 2012 at Very Large Database Conference in Istanbul: "In terms of scalability, there is a clear winner throughout our experiments. Cassandra achieves the highest throughput for the maximum number of nodes in all experiments with [linearly] increasing throughput from 1 to 12 nodes [21].

4.5.3 AVAILABILITY & COMMUNITY

Cassandra does not have any official forum for its users. The official web-page provides a mailing list [http://www.mail-archive.com/user@cassandra.apache.org/], with the hottest topics. However, the format is not modern and the navigation is not satisfying. In addition with this datastrax offers a big variety of tutorials regarding Cassandra [http://academy.datastax.com/] and for problem-solving questions, the popular StackOverFlow [http://stackoverflow.com/] can be an option.

Regarding third party libraries, Apache Cassandra supports the following programming languages:

- C#
- C++
- Clojure
- Erlang
- Go
- Haskell
- Java
- JavaScript
- Perl
- PHP
- Python
- Ruby
- Scala

4.6 COMPARISON

In this chapter an attempt is made to identify which document store fulfills better the most common requirements according to developers.

4.6.1 QUERY LANGUAGE

Regarding, query language, two sets can be introduced. The former contains Azure DocumentDB and Apache Cassandra and the latter contains MongoDB. The document stores of the first set let user to use a subset of SQL when there is a need of query. This feature is sometimes crucial because many developers feel more familiar with a new technology, when this technology uses aspects from the previous one. As a result, a data-scientist with a basic SQL background can start using Azure DocumentDB and Apache Cassandra immediately. On the other hand, MongoDB has its own query language which is different in terms of syntax than the well-known SQL syntax. However, a developer who can implement basic SQL queries as the examples above, would be able to start querying with not so much effort.

4.6.2 SUPPORTED DRIVERS

According to supported drivers and programming languages, MongoDB first and then Cassandra, consist of an ideal option. Unfortunately, and because of the fact that Azure DocumentDB is not an open-source software, the supported languages are limited. As a result, MongoDB and Cassandra are a step ahead because they support a big amount of programming language, but practically this step is not so big, since Azure DocumentDB supports the most used ones.

4.6.3 SUPPORT

Another metric, which can be taken into account is the support that each store offers to its users. According to this aspect, MongoDB is the number one since it is the most popular document store in database community. After MongoDB, the most popular document store and the number eight of the most popular databases is Apache Cassandra [21]. Therefore, using the axiom which claims "the more users the better support" MongoDB and Apache Cassandra take lead against Azure DocumentDB. However, in recent times, Microsoft is a leader company and its commercial use products come with professional support.

4.6.4 SUMMARY

To sum up, if we would like make a conclusion, this would be that MongoDB is the most popular document store which is free, open-source, with great support regarding programming languages and huge community but with a different than SQL query language. As a result, it consists of a good start for everyone who wants to introduce himself to document stores. A good alternative, would be Apache Cassandra, which has the extra feature of using a subset of SQL as query language. For developers, who prefer to rely on giants, Microsoft's Azure DocumentDB would be a option with no risk.

5 STREAM STORES

The second kind of stores that is introduced refers to stream. Stream stores look like more to real-time business systems than to regular database systems. Since the previous decade, computer science community was concerning about a kind of system, which can support business intelligence tasks and perform analytics in real time. However, the existent technology was limiting every attempt of implementing this kind of systems. More specifically, these data-warehousing environments that were offering high performance were also characterized by high cost. On the other hand low budget systems were characterized by extreme low performance and high latencies.

Nowadays, an amount of handy open source platforms have brought a new dawn on real-time analytics. Two of the most popular systems are **Apache Storm** and **Apache Spark**. Both of these stream stores, which consist of projects within the Apache Software Foundation, support real-time processing capabilities to a much wider range of potential users [25]. At the moment that this research takes place, Apache Spark is a full, top level Apache project. On the other hand, Apache Storm consists of a newborn project.

Both Apache Spark and Apache Storm are implemented in JVM based languages. More specifically, the former is implemented in Scala (functional and object-oriented language) and the latter in Clojure (a dialect of Lisp targeting the JVM providing the Lisp philosophy).In this chapter an attempt is made to investigate, how these two tools provide overlapping capabilities and at the same time each of them has distinctive features and structure [28].

5.1 APACHE STORM

As it mentioned in introduction 5, a revolution in data processing takes effect, in recent times. The existence technology has made possible the storage and the process on data at scales that during earlier years were unreachable. However, these data processing technologies are not real-time systems, nor are they meant to be. The reason is that a real-time data processing occurs with a fundamentally different set of requirements than batch processing.

On the other hand, real-time data processing at massive scale arises more and more as a requirement in business world. Apache Storm created to fulfill this requirement [27]. "Apache Storm is a distributed, fault-tolerant, open-source computation system that allows you to process data in real-time with Hadoop. Storm solutions can also provide guaranteed processing of data, with the ability to replay data that was not successfully processed the first time" [29].

In other words, Apache Storm is a real-time computational engine. A real-time computational engine which is open-source and works simple. More specifically, Apache Storm runs continuously, consuming data from configured sources, which are named Spouts. These data are passed down to the processing pipelines, which are named Bolts. The combination of Spouts and Bolts consists of a Topology [28].

5.1.1 ARCHITECTURE

The architecture of Apache Storm is based on a master/slave architecture. More specifically, it consists of three unique types of nodes and each of them function differently in order to process messages with guarantee [30]. These three nodes are introduced below:

- ***Nimbus***: Nimbus consists of the master node. More specifically, Nimbus is a daemon process, which is responsible for the deployment, managing, coordinating and monitoring all the topologies that are executed on the cluster. One of Nimbus node basic tasks is to assign and reassign the entire work when a failure occur. Moreover, this node is responsible for the distribution of any required file to a supervisor node [30].
- ***ZooKeepers***: ZooKeepers are the node, which connects Nimbus (master node) to Supervisors (worker nodes). Taking into account that in this distributed environment such as Apache Storm, centralized events and information coordinates are critical for processes coordination. Apache ZooKeeper functions have the task of storing state's information such as assignments of work and job statuses between the master node and the worker nodes [30].
- ***Supervisors***: As it mentioned above, Supervisors consist of the worker-nodes. More specifically, it is a daemon process that "spawns" new tasks and monitors the status of the workers [30].

These three components collaborate as it is presented in Figure 5.1.

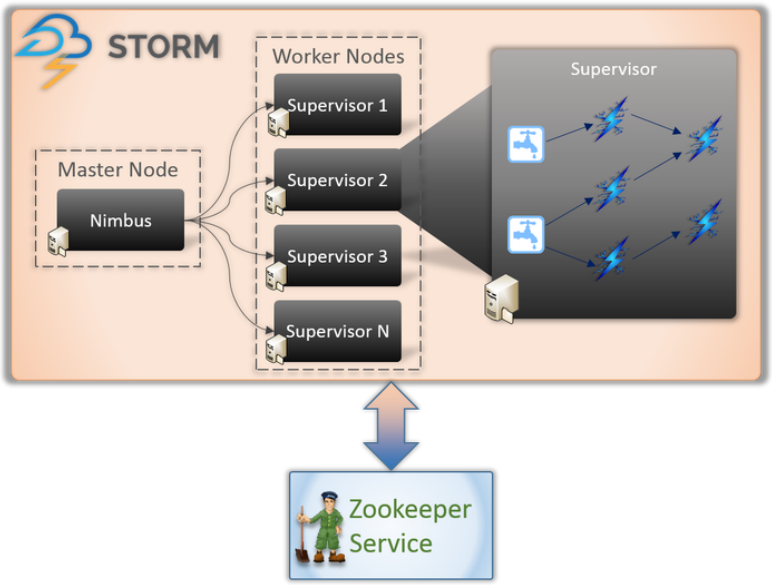


Figure 5.1: The architecture of Apache Storm [30]

In order to start doing real-time computation on Storm, we need to create "topologies". A topology can be defined as a graph of computation. Inside a topology, every node contains processing logic, and is linked with other nodes. These links between the nodes indicate the way that data should be passed fro the one node to the other. All these components such us the topologies and the nodes are essential to understand Apache Storm. For this purpose all the terminology that is required is introduced below:

Topologies A simple Storm topology is illustrated, showing Spouts (referring to data sources) and Bolts (referring to nodes related to stream processing logic). A Storm topology could similar to a job of the traditional or batch processing systems. In order to identify the structure of a topology in Storm, we can claim that each topology is made up of data streams, spouts and bolts. These topologies are deployed to and run by an Apache Storm cluster [30].

Data Streams As data stream, a flow of data is defined. More specifically, a data stream is an unbound stream of information (data) in a from of a collection of key/value pairs. These data are the most basic data structure within storm [30] [32].

Spouts A spout is the entry point into a Storm topology. More specifically, it connects to a source of data, then transforms the contained data into a tuple and at the end emits this tuple for consumption and processing by one or more bolts [30] [32]. It is represented by the water faucet in Figure 5.1.

Bolts A bolt is responsible for the transformation and the process of data within a topology. Bolts receive data streams from spouts or other bolts as an input and they output none or more data streams. Bolts process data and execute tasks such as calculations, aggregations, filtering, joins and writes to external sources [30] [32]. It is represented by the blue thunder in Figure 5.1.

Table 5.1 summarizes each component of Apache-Storm Architecture.

5.1.2 SCALABILITY

As a stream store, Storm can scale to a massive number of messages per second. In order to achieve scale in a topology, two tasks must be done. The former is to increase the machines of this topology and the latter to increase the parallelism settings of this specific topology, as well. For instance, an example of Storm's scale is following: one of Storm's initial applications processed a million messages per second on a 10 node cluster, including hundreds of database calls per second as part of the topology. Storm's usage of Zookeeper for cluster coordination makes it scale to much larger cluster sizes [30].

Table 5.1: Storm Concepts [31]

Storm Concept	Description
Tuple	A named list of values of any data type. The native data,structure used by Storm.
Stream	An unbounded sequence of tuples.
Spout	Generates a stream from a real-time data source.
Bolt	Contains data processing, persistence, and messaging alert logic. Can also emit tuples for downstream bolts.
Stream Grouping	Controls the routing of tuples to bolts for processing.
Topology	A Storm application. A group of spouts and bolts wired together into a workflow.
Processing Reliability	Storm guarantee about the delivery of tuples in a topology.
Workers	A Storm process. A worker may run one or more executors.
Executors	A Storm thread launched by a Storm worker. An executor may run one or more tasks.
Tasks	A Storm job from a spout or bolt.
Process Controller	Monitors and restarts failed Storm processes.
Master/Nimbus Node	The host in a multi-node Storm cluster that runs a process controller. The process controller is responsible for restarting failed process controller daemons, such as supervisor on slave nodes. The Storm nimbus daemon is responsible for monitoring the Storm cluster and assigning tasks to slave nodes for execution.
Slave Node	A host in a multi-node Storm cluster that runs a process controller daemon, such as supervisor, as well as the worker processes that run Storm topologies. The process controller daemon is responsible for restarting failed worker processes.

5.1.3 STORM USERS

Apache Storm users do not have an official form but a mailing list. A user in order to send messages to this list should send an email to user@storm.apache.org. Subscription is done by sending an email to user-subscribe@storm.apache.org and the canceling of this is done by sending an email to user-unsubscribe@storm.apache.org. In addition to with the online support are arranged. Some of them take place in London, Boston and New York [27]. A topology can be written in any language including any JVM based language, Python, Ruby, Perl, or, with some work, even C.

5.2 APACHE SPARK

Apache Spark is a cluster computing platform designed to be fast and general-purpose. On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large data-sets, as it means the difference between exploring data interactively and waiting minutes or hours. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk [33].

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools [33].

Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries. It also integrates closely with other Big Data tools. In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra [33].

5.2.1 ARCHITECTURE

The Spark project contains multiple closely integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a computing cluster. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to inter-operate closely, letting you combine them like libraries in a software project [33].

A philosophy of tight integration has several benefits. First, all libraries and higher-level components in the stack benefit from improvements at the lower layers. For example, when Sparkśore engine adds an optimization, SQL and machine learning libraries automatically speed up as well. Second, the costs associated with running the stack are minimized, because instead of running 510 independent software systems, an organization needs to run only one. These costs include deployment, maintenance, testing, support, and others. This also means that each time a new component is added to the Spark stack, every organization that uses Spark will immediately be able to try this new component. This changes the cost of trying out a new type of data analysis from downloading, deploying, and learning a new software project to upgrading Spark [33].

Finally, one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously, analysts can query the resulting data, also in real time, via SQL (e.g., to join the data with unstructured log-files). In addition, more sophisticated

data engineers and data scientists can access the same data via the Python shell for ad-hoc analysis. Others might access the data in standalone batch applications. All the while, the IT team has to maintain only one system [33].

Here we will briefly introduce each of Sparkś components, shown in Figure

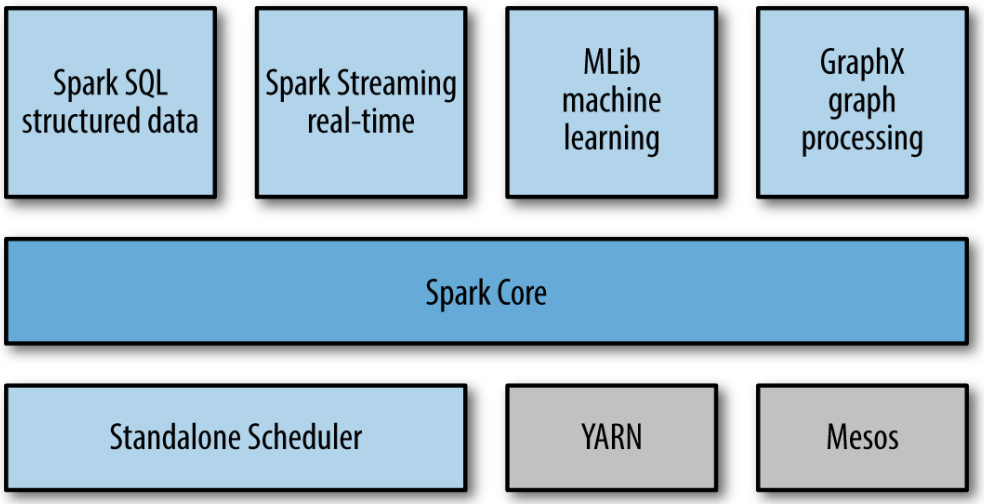


Figure 5.2: The basic components of Apache Spark service [33].

The components of Figure 5.2 are described below:

Spark Core Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed data-sets (RDDs), which are Sparkś main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections [33] [34].

Spark SQL Spark SQL is Sparkś package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL called the Hive Query Language (HQL) and it supports many sources of data, including Hive tables, Parquet, and JSON. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics. This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0. Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs [33] [34].

Spark Streaming Spark Streaming is a Spark component that enables processing of live streams of data. Examples of data streams include log-files generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming provides an API for manipulating data streams that closely matches the Spark Coreś RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time. Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core [33] [34].

MLlib Spark comes with a library containing common machine learning (ML) functionality, called MLlib. MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster [33] [34].

GraphX GraphX is a library for manipulating graphs (e.g. a social networkś friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., sub-graph and map-Vertices) and a library of common graph algorithms (e.g. PageRank and triangle counting) [33] [34].

Cluster Managers Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler. If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; if you already have a Hadoop YARN or Mesos cluster, however, Sparkś support for these cluster managers allows your applications to also run on them [33] [34].

5.2.2 Spark Users

Because Spark is a general-purpose framework for cluster computing, it is used for a diverse range of applications. However, two major groups can be defined; the data scientists and the engineers. Of course, these are imprecise disciplines and usage patterns, and many folks have skills from both, sometimes playing the role of the investigating data scientist, and then âĂIJchanging hatsâĂİ and writing a hardened data processing application. Nonetheless, it can be illuminating to consider the two groups and their respective use cases separately. The official Spark community can be found at https://spark.apache.org/community.html. As Storm, Spark offers a mailing list for its users. The following email addresses offered [36]:

- user@spark.apache.org is for usage questions, help, and announcements.
- dev@spark.apache.org is for people who want to contribute code to Spark.

5.3 COMPARISON

Comparing these two systems, if the requirements are primarily focused on stream processing and CEP-style processing regarding a greenfield project with a purpose-built cluster for the project, then Storm is probably the choice - especially when existing Storm spouts that match integration requirements are available. This is by no means a hard and fast rule, but such factors would at least suggest beginning with Storm. On the other hand, if the case is the leveraging of an existing Hadoop or Mesos cluster and/or if this process needs involve substantial requirements for graph processing, SQL access, or batch processing, then Spark could be the first step.

Another factor to consider is the multilingual support of the two systems. For example, if there is a need to leverage code written in R or any other language not initially supported by Spark, then Storm has the advantage of broader language support. By the same token, if there is a requirement of having an interactive shell for data exploration using API calls, then Spark offers a feature that Storm does not.

To sum up, usually users want to perform a detailed analysis of both platforms before making a final decision. Because of the fact that these platforms are similar, it would be useful, to build a small proof of concept - then run their own benchmarks with a workload that mirrors they anticipated workloads as closely as possible before fully committing to either. On the other hand, of course, a user does not need to make an "either/or" decision. Depending on the workloads, infrastructure, and requirements, a technical user may find that the ideal solution is a mixture of Storm and Spark - along with other tools like Kafka, Hadoop, Flume, and so on. Therein lies the beauty of open source.

6 GRAPH STORES

The third and last kind of storage, which is examined refers to a graph databases. More specifically, a graph database, also called a graph-oriented database, is a type of NoSQL database that uses graph structures for semantic queries with nodes, edges and properties in order to represent and store data.

A graph database is consisted of by a collection of nodes and edges. Trying to make a parallelism with the relational databases schema, each node represents an entity and each edge represents a relationship between two nodes. Consequently, every node, which is defined by a unique identifier, a set of properties expressed as key/value pairs and a set of outgoing edges and/or incoming edges. Regarding the edges, each edge is defined by a unique identifier,a direction, a type, a start-node, an end-node and a set of properties. These aspects can be seen in Figure 6.1:

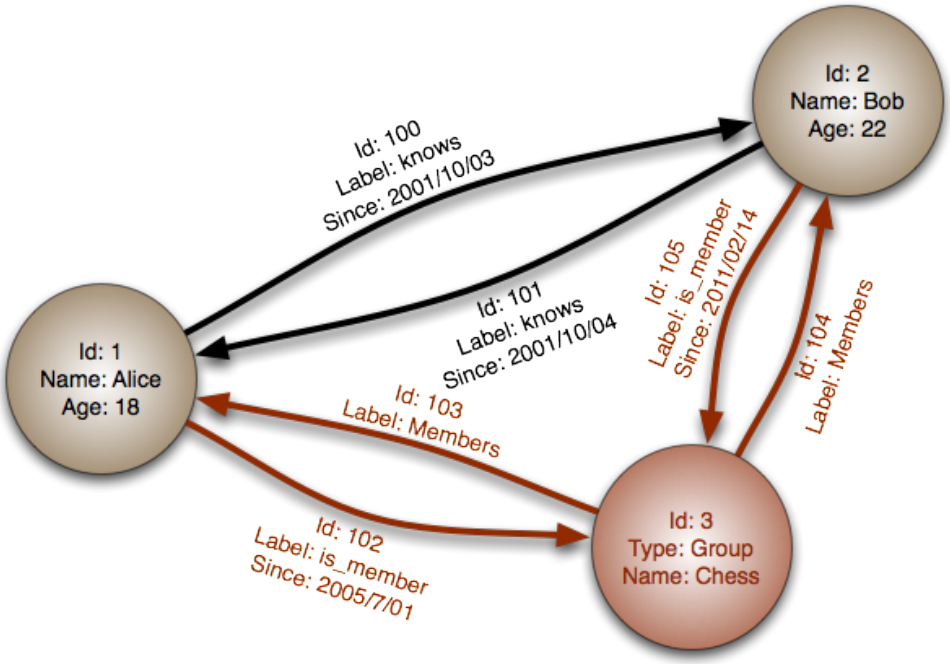


Figure 6.1: An example of a graph structure schema 6.1

In this example, it is obvious the fact that there are two nodes that represent the students Alice and Bob. Each student, has his/her attributes: id, name and age inside his/her node. In addition with Alice and Bob, there is one more node which represents a student group for students who play chess. A significant difference between a graph store and a document store is the fact that the graph stores focus on the relationship between the nodes. More specifically, the edges that represent the relationship unravel information regarding the nodes. As a result, in the above example, a user could search for all the nodes(people) that *"Bob" "knows"* after the date 02/10/2001 or for all the nodes(groups) that *"Bob" "is_member"*.

6.1 SUITABLE CASES

Graph databases are suitable for 3 kind of cases:

Connected Data Graph databases are ideal for interconnection analysis. Social media are based on connections among the users, which can be represented efficient by a graph database, offering the ability of data mining tasks. In general, graph databases are useful for working with data in which is characterized by complex relationships and dynamic schemas.

Routed, Dispatch and Location-Based Services Considering the "Traveling salesman problem" [37], graph databases can assist efficiently with problems where the data can have the form of a location or an address (node) and the relationship edges can represent a distance. Distance and location attributes can be used in order to build a recommendation system.

Recommendation Engines The connected graph, which is created by the data can be used to express the association between a client and a potential product. More specifically, regarding its connections, a node can be an ideal client for a product which its neighbors use, in terms of preference similarity. In this manner, movie, book and restaurant recommendation systems, which offer accurate suggestions to their users, can be built.

6.2 EXAMINED DATABASES SYSTEMS

In the following sub-chapters, three document stores are introduced:

- **Neo4j**
- **OrientDB**

6.3 NEO4J

Neo4j is an open-source graph database implemented in Java, by Neo Technology. The first release of Neo4j became available in 2007 [38]. The official website claims that Neo4j is "the World's Leading Graph Database" because of its nature. More specifically, they support that graphs are the ideal way to work with data, since they can express efficiently how the human mind works [39]. The investigated Neo4j release is the 2.3.0, October 2015.

6.3.1 QUERY LANGUAGE

The query language of Neo4j is named Cypher. Cypher is defined as declarative, SQL-inspired language which describes concepts and patterns in a graph database. More specifically, Cypher provides the common actions: select, insert, update and delete from a graph database, without requiring from the user to declare how to do it [40].

Nodes With Cypher ASCII-Art is used for the representation of the patterns. Precisely, nodes are surrounded with parentheses. These parentheses look like the circles of the graph

schema. Each node can have an identifier for reference purposes. For instance, (p) could be a identifier for people and (t) for things. However, it is recommended that the identifier name should be long and more expressive for real-world queries, such us (person) or (thing).

An example could be "find all the people and the things they like". In this case the query would include the representative identifiers for the people and the things:

$$(p)->(t)$$

The properties of the nodes can be accessed with the use of the ., such us $p.name$ or $thing.color$. The general structure is:

$$MATCH\ (node)\ RETURN\ node.property$$

$$MATCH\ (node1)->(node2)$$
$$RETURN\ node2.propertyA, node2.propertyB$$

Relationships Relationships provide additional information regarding the connections between the nodes. In the previous case, a node could be a person (p), but it may represent a supplier or a seller of a thing (t). As a result, relationships describe how the nodes are related with each other.

In Cypher, a possible query could be "retrieve everyone who likes a thing". This pattern could be described as

$$(p)-[:LIKE]->(t)$$

and it retrieves all the nodes type of (p) that have a relationship typed LIKE with other nodes type of (t).

Or generally:

$$MATCH\ (node1)-[:REL\_TYPE]->(node2)$$

As in nodes, the properties of the relationships can be accessed with the use of identifiers for relationships (in front of the $:TYPE$). For example, in order to access the information about a relationship such as a rating that a person gave to a thing about how much he likes it, could be

$$MATCH\ (p)-[how:LIKE]->(t)$$

The general structure is

$$MATCH\ (node1)-[rel:TYPE]->(node2)$$

$$RETURN\ rel.property$$

Labels Labels are like stamps, which assign a role or a type to the nodes. So, with the use of labels, for instance, people can be distinguished from companies. More specifically, by matching for $(p:Person)-[:LIKE]->(t)$, the database will return "John" and/or "Anne" for example but not "Hasbro"-a well know company. Label general structure looks like:

$$MATCH\ (node:Label)\ RETURN\ node$$

$$MATCH\ (node:Label)\ WHERE\ node.property = "value"\ RETURN node$$
$$MATCH\ (node1:Label1)-[:REL\_TYPE]->(node2:Label2)$$

$$RETURN\ node1,\ node2$$

6.3.2 SCALABILITY

In this section, the features that introduce Neo4j as a scalable database, in terms of horizontal scalability, are described. The features are the following [42]:

Scalable Distribution Architecture This section indicates the need of availability to horizontally scale a database's records across multiple nodes. In Neo4j, the only option which is supported is to replicate the complete database only. This feature which restricts scalability to the capacity of a single node. More details can be found in The Neo4j Manual v2.3.1 - Chapter 25 "High Availability".

Scaling Out - Adding Data Storage Capacity An important feature of a scalable database is the ability to add new nodes into a cluster. More specifically, in a ideal database this feature should be triggered automatically by the database, with the minimum overhead and zero downtime. Consistent hashing (hash functions) is designed to minimize data movement as capacity is scaled up (or down), and generally databases that support consistent hashing will be able to utilize new resources with minimal data movement. Databases that require significant administration to add capacity, or must be taken offline, are likely to be much harder to scale and less available. In Neo4j, re-balancing of data is: N/A - single server only. Details can in The Neo4j Manual v2.3.1 - Chapter 1 "Neo4j Highlights".

Request Load Balancing Load balancing of client requests is a fundamental scalability mechanism for evenly spreading request load across all available database resources. If a database only provides static, fixed connections to its request handling processes, this can inhibit scalability as client requests cannot be easily spread across database resources, leading to hot spots. Load balancing approaches are either (1) client-based, with each clients holding a pool of connections which it selects from using an approach such as round-robin, or (2) server-based, when an intermediary process in the database intercepts client requests and load-balances them across replicated resources. Server-based approaches can introduce bottlenecks if there are only a small,

restricted size pool of load balancing processes possible. In Neo4j, the following capability is provided: uses HTTP-based load balancers. Details can be found The Neo4j Manual v2.3.125.8. - "Setting up HAProxy as a load balancer".

Granularity of Write Locks The granularity of write locks is an important scalability feature. Ideally only the data objects being updated by a request should be locked using a fine grain locking algorithm. In some databases, coarser database 'page' or 'collection' level locking can severely impede the scalability of write loads. In Neo4j, data object-level write locks are locks on updated objects only. Details can be found in The Neo4j Manual v2.3.118.3. - "Default locking behavior"

Scalable Request Processing Architecture Bottlenecks in the request processing path for reads and writes can rapidly become inhibitors for scalability in a big data system as concurrent request loads increase. These bottlenecks are typically request or transaction coordinators that cannot be distributed and replicated, or processes that store configuration state that must be accessed frequently during request processing. In Neo4j, the request processing path is based on an external load balancer. Details can be found The Neo4j Manual v2.3.125.8. - "Setting up HAProxy as a load balancer" [43].

6.3.3 AVAILABILITY & COMMUNITY

"Neo4j has the largest and most vibrant graph database community in the world. This release would not have been possible without a continuous stream of input from community members. Neo4j 2.2 represents our largest beta to date, with participation from more than two thousand users" claimed Philip Rathle (Neo Technology's VP of Products) [41]. Neo4j supports its users via

- Stackoverflow
- Google groups
- Licence customer support

Moreover there plenty of unofficial groups and forums, about Neo4j issues.

Regarding third party libraries, Neo4j supports the following programming languages [**?**]:

- Java
- .NET
- JavaScript
- Python
- Ruby
- PHP
- R
- Go
- Clojure
- Perl
- Haskell

6.4 ORIENTDB

OrientDB is the second graph store, which is examined during this research. It is an open source database that is written in Java and belongs to "Second Generation Distributed Graph Database Systems". The main difference between First and Second generation Graph Databases is the lack of features regarding Big Data requirements, such as multi-master replication, sharing. In addition, OrientDB offers the flexibility of Documents. More specifically, it internally manages relationships like a traditional graph databases but can work also in schema-less mode like a document database. As a result, OrientDB is a graph and a document database at the same time [44].

As the official web-page claim "OrientDB is incredibly fast". More specifically, developers advocate that "it can store around 220,000 records per second on a common hardware. Even for a Document based database, the relationships are managed as in Graph Databases with direct connections among records. You can traverse parts of or entire trees and graphs of records in a few milliseconds. Supports schema-less, schema-full and schema-mixed modes. Has a strong security profiling system based on user and roles and supports SQL among the query languages. Thanks to the SQL layer, it's straightforward to use for those skilled in the relational database world" [44].

6.4.1 QUERY LANGUAGE

Users of OrientDB use an extended subset of SQL, in order to query information from the database. As a result, instead of inventing an new query language, OrientDB querying takes place with the widely used and well-understood language of SQL. Of course, the query language of OrientDB is an extended version of SQL in order to support more complex graphing concepts, such as Trees and Graphs. The fundamental concepts are introduced below:

SELECT The SELECT statement queries the database and returns results that match the given parameters. For instance, earlier in Getting Started, two queries were presented that gave the same results: BROWSE CLUSTER user and BROWSE CLASS User. Here is a third option, available through a SELECT statement.

SELECT FROM User

Notice that the query has no projections. This means that you do not need to enter a character to indicate that the query should return the entire record, such as the asterisk in the Relational model, (that is, SELECT * FROM User).

Additionally, User is a class. By default, OrientDB executes queries against classes. Targets can also be:

Clusters To execute against a cluster, rather than a class, prefix CLUSTER to the target name.

SELECT FROM CLUSTER : *User*

Record ID To execute against one or more Record ID's, use the identifier(s) as your target. For example:

$$SELECT\ FROM\ \#103$$
$$SELECT\ FROM\ [\#10:1, \#10:30, \#10:5]$$

Indexes To execute a query against an index, prefix INDEX to the target name.

$$SELECT\ VALUE\ FROM\ INDEX:dictionary\ WHERE\ key='Jay'$$

WHERE Much like the standard implementation of SQL, OrientDB supports WHERE conditions to filter the returning records too. For example,

$$SELECT\ FROM\ User\ WHERE\ name\ LIKE\ 'l\%'$$

This returns all User records where the name begins with l

6.4.2 SCALABILITY

OrientDB has the ability to be distributed across different servers and to be used in a different manner in order to achieve the maximum of its performance, scalability and robustness. OrientDB uses the open source "Hazelcast Open Source project" for clustering management. For more information click on https://hazelcast.com/ OrientDB has a Multi-Master + Shared architecture. As a result, all the servers are master and because of this feature horizontal scalability and reliability are provided. More specifically:

Elastic Linear Scalability The master-slave technique often has a bottleneck. More specifically, because of the fact there is only one master for a lot of slaves and because of the fact that there is a big difference related to the entry point of data on master and slave, while the master has multiple simultaneously threads inserting/updating/deleting data, on the slave is just a single one responsible for deal with all these transactions. By using OrientDB, throught put is not limited by a single server but there is a global throughput, which is the sum of the all servers' throughput. Thus, in case there is a need to scale up, a server node can be simply added inside the network. This server will automatically join the existing distributed server cluster with zero configuration. As well with the join, synchronization occurs automatically as soon as the server is online [45].

"Practically, this means that the transactional engine that can run in distributed systems supporting up to 302,231,454,903,657 billion (2^{78}) records for maximum capacity of 19,807,040,628,566 petabytes of data distributed on multiple disks in multiple nodes" [45].

Distributed Reliability Most NoSQL solutions are used as "cache" to speed up certain use cases, while the master database remains a relational DBMS. For this reason, the average NoSQL product is built more for performance and scalability, while sacrificing reliability [45].

Perfect Cloud Database Solution According to the aforementioned benefits, OrientDB is an ideal option for the Cloud. Hundreds of servers can split the workload, scaling horizontally across distributed servers or data centers [45].

6.4.3 AVAILABILITY & COMMUNITY

OrientDB is offered in two editions. The former is the "Community Edition" and is free for any use (Apache 2 license). The latter is named "Enterprise Edition" and is a product for commercial use. Development of OrientDB Community Edition is supported by its users. OrientDB User Community with more than one hundred contributors and more than three thousand users world-wide.

OrientDB supports 3 kinds of drivers [46]:

- **Native binary remote**, which communicates directly with the TCP/IP socket using the binary protocol
- **HTTP REST/JSON**, which communicates directly with the TCP/IP socket using the HTTP protocol
- **Java wrapped**, as a layer, which links in some way the native Java driver. This is pretty easy for languages that run into the JVM like Scala, Groovy and JRuby

In General OrientDB supports:

- Java (native) API
- JDBC driver
- NodeJS
- PHP
- .NET
- Python
- C
- Javascript
- Ruby
- Groovy
- Scala
- R
- Elixir
- Clojure
- OrientDB
- Perl

6.5 COMPARISON

In this chapter, an attempt is made to identify which graph store is more appropriate according to the most common requirements.

6.5.1 QUERY LANGUAGE

Regarding, query language, the two stores are totally different. Neo4j introduces the user to "Cypher"- its very own query language. As a result, a few or more training is required in order to learn and understand the new language. On the other hand, OrientDB uses a query language, which is built like SQL, with some additional plugins to operate on graphs. For instance, considering there is a need to retrieve the name of Alice and all the groups that participates in:

OrientDB

$$SELECT\ Name,\ out('is\_member').Name$$
$$FROM\ Person$$
$$WHERE\ Name = 'Alice'$$

Neo4j

$$MATCH\ (student:Person\{name:'Alice'\}) - [:is\_member] - > (group)$$
$$RETURN\ student.name,\ group.name$$

As a result data-scientists with SQL background find OrientDB more familiar than Neo4j.

6.5.2 SCALABILITY

Both of the stores -Neo4j and OrientDB- provide replication. The difference between Neo4j and OrientDB is the fact that Neo4j uses a Single Master-Slave architecture: only one server can be the master, at any moment. On the other hand, OrientDB offers Multiple Master architecture: all the servers are masters. As a result, OrientDB support linear scalability but Neo4j not. This feature can be a huge potential bottleneck on write operations, since Neo4j would not be able to scale.

6.5.3 SUPPORT

According to support that each store offers to its community, Neo4j has the advantage. Since it is one of the most popular NoSQL stores, a lot of groups work on this and as a result, it expands fast in terms of supported languages and forums. More specifically, Stackoverflow is the main forum for Neo4j users. However, OrientDB offers also a big variety of language drivers but its community is smaller than Neo4j.

6.5.4 OPERATIONAL DBMS

As indicates the name, NoSQL solutions, are usually used as a Not-only SQL component. For instance, they exist as a "cache", in order to utilize specific cases or demands, while the main database-system is still a relational DBMS. Thus, a lot NoSQL stores focus on performance and scalability, instead of reliability. This can conclude that industry or scientific community do not consider NoSQL DBMSs as trustworthy as Relational DBMSs. However, OrientDB, by using a Write Ahead Logging technique, is able to restore a database after a crash. As a result, it fulfills the ACID set properties and can roll back transactions [47]. This feature makes OrientDB more interesting that Neo4j and the rest of NoSQL DBMSs.

6.5.5 SUMMARY

To sum up, if we would like make a conclusion, this would be that Neo4j is more popular than OrientDB. Both of them offer great support regarding programming languages and huge

community. The fact that OrientDB has a query language similar to SQL makes it more appropriate for a beginner. As a result, OrientDB could consist of a good start for everyone who wants to introduce himself to graph stores. On the other hand, the bigger community of Neo4j can offer more tutorials or support to a new user. Regarding, scalability OrientDB has a slight advantage against Neo4j, but the purpose of our project might not use this extra feature. In addition, the ACID behavior or OrientDB, could be considered as an extra feature that some users could pay for it.

7 NO NOSQL DATABASES

In contrast with this study, there is a trend, which advocates that, sometimes, sticking with the default may be the best solution for a data management problem. Nowadays, NoSQL databases introduce themselves as the new trend in data management, by adding new features in storage logic and by solving a lot of classic data management problems. However, in many cases the traditional relational databases seems to be the ideal solution.

The advantages when a relational database is used are the following:

Well-known Each computer and data scientist is trained and aware of using a relational database. More specifically, they know the fundamentals of a typical SQL database and they can use it with almost no training in the most of the cases.

Mature Relational databases have already passed the rough tests of a new technology. There are plenty of drivers and tools, which can connect them with the rest of the applications and communities which can support the users. Moreover, there are no political issues which should be taken into account regarding their use.

Still popular Taking into account that Facebook in year 2015 still uses a SQL database for the storage of millions of records, we understand that a relational database can still support the needs of an application, which intends to deal with big data.

As a result, according the problem and no matter if there is a NoSQL database, which looks ideal for it, it's necessary to investigate if the use of a relational database looks more promising than the NoSQL one. As in every aspect in life, there in no solution which fulfills all the obligations and especially in computer science "world", a technology never treats every problem in a perfect way. Thus, before think about NoSQL databases we have to accept the fact that the most of the times the best solution could be a simple SQL database, with all the pros and cons that deliver.

8 EXPERIMENTAL PART

According to Chapter 7, SQL databases still consist of the ideal solution for a lot of data management problems. However, this report advocates that NoSQL stores are alternatives that can "replace" SQL stores, in problems with specific demands. This section illustrates an experiment which took place, in order to specify which solution fits better to the examined data-set, which is introduced in Chapter 3.

8.1 SCENARIO

The scenario of the experiment focus on two major aspects. The former is time-line representation and the latter is performance analysis. According to these aspects, a set of benchmark queries, as been built, which cover the a set of requirements regarding querying the input data-set.

8.2 EXPERIMENT SET-UP

This specific experiment will focus on MongoDB and MySQL. The reason that these two databases are chosen is their popularity. MongoDB is the most popular NoSQL store and MySQL is the most popular non-commercial SQL store according to DB-Engines. More specifically, these two data stores have been filled-up with a soccer video analysis data-set and then a set of benchmark queries has been applied to these specific data stores.

8.2.1 BENCHMARK QUERIES

As it mentioned above, the set of the benchmark queries focuses on the time-line representation of the data-set and on the performance analysis. More specifically, there are seven queries, which operate on the input data-set. These queries are the following:

1. Project the position of a player(s) in the field during the time
2. Project the position of the whole team in the field during the time
3. Project which players overlap their teammates positions
4. Project the average speed of the whole team
5. Project the median position of the team (formation) in the field

8.3 IMPLEMENTATION

The two data stores have been implemented by their DBMS for Linux. The system info are the following:

OS Linux 4.4.0-28-generic, Ubuntu 16.04 LTS

CPU 4x Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz (862.589MHz)

RAM 11.72GB

Regarding the data-stores, the following platforms have been installed:

MongoDB MongoDB shell version 2.4 and Robomongo have been installed. Robomongo is a shell-centric cross-platform open source MongoDB management tool (i.e. Admin GUI). Robomongo embeds the same JavaScript engine that powers MongoDB's shell.

MySQL MySQL 5.7.12 and Workbench 6.3 have been installed.

8.3.1 DATA-SET IMPORT

The data-set, which is described in Chapter 3, has been imported in MongoDB and MySQL database systems. In MongoDB database, the input data-set has been imported via Shell commands, which are described by Figure 8.1.

```
$mongoimport -d Tromso -c tromso_anji -type csv -file tromso_anji_first.csv -headerline
Connected to: 127.0.0.1
Thu Jun 30 15:51:50.029 Progress: 16897601/50960343 33%
Thu Jun 30 15:51:50.029 118100 39366/second
Thu Jun 30 15:51:53.006 247700 41283/second
Thu Jun 30 15:51:55.432 check 9 354484
Thu Jun 30 15:51:56.040 imported 354483 objects

$mongoimport -d Tromso -c tromso_anji -type csv -file tromso_anji_second.csv -headerline
Connected to: 127.0.0.1
Thu Jun 30 15:52:20.006 Progress: 14903023/44253536 33%
Thu Jun 30 15:52:20.006 120600 40200/second
Thu Jun 30 15:52:23.158 250500 41750/second
Thu Jun 30 15:52:25.918 check 9 366162
Thu Jun 30 15:52:26.594 imported 366161 objects
```

Figure 8.1: Console of MongoDB import process

```
File tromso_anji_first.csv was imported in 1670.120s
Table tromso.tromso_anji has been used
354483 records imported

File tromso_anji_second.csv was imported in 1810.970s
Table tromso.tromso_anji has been used
366161 records imported
```

Figure 8.2: Console of MySQL import process

On the other hand, in MySQL database, the input data-set has been imported via Workbench bulk functions. More specifically, Workbench offers csv import function, which utilize

the import queries and optimize the input process. The output of Workbench is described by Figure 8.2.

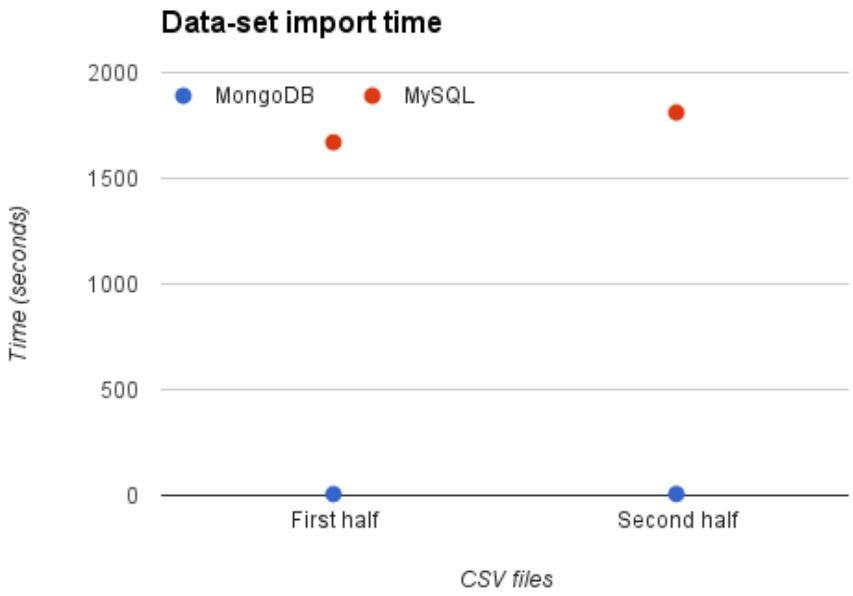


Figure 8.3: The data-set import performance of each data store in terms of time.

By inspecting the time that these two database systems, it is concluded that MongoDB with its schema-less structure achieves to be faster than MySQL. More specifically, MongoDB is around 280 times faster than MySQL and as it is presented by Figure 8.3, in both of csv files, MySQL needed 28 and 30 minutes respectively, when MongoDB only 6 seconds.

However, this huge might be explained by the fact that MongoDB shell commands are lighter than a whole GUI application which requires sources and time in order to optimize and transform a csv file into a relational table. On the other hand, this is a disadvantage of relational model. Moreover, the input csv files should be modified in order to compromise with the structure of the relational table. Precisely, there was a need to add an extra column, which indicates the primary key of the records, inside the csv file. This transformation was done with the use of Microsoft Ecxel application and also consumed a considerable amount of time. On the other hand, MongoDB behaves with high flexibility and speed.

8.4 RESULTS

In this section the results of the experiment are presented. Sub-Chapter 8.4.1 is an introduction, which illustrates all the aspects, which have been taken into account regarding experiment execution. Each of the following subsection refers to one benchmark query of Section 8.2.1 and presents the results via tables and charts.

8.4.1 ASSUMPTIONS

The experiment has five phases. Every phase refers to one specific benchmark query. More specifically, during each phase a specific query executed in both of the data-stores. Each phase consists of ten query executions for each data-store. More specifically, all the experiments in MySQL are done via Workbench platform and all the experiments in MongoDB via Robomongo application. Something that must be mention is that Workbench calculates the duration of each query as a summary of $Query\,duration + Fetching\,time$ and Robomongo calculates the $Query\,duration$ as a whole. Because of that, there are two bars in each chart regarding MySQL: one with $Fetching\,time$ and one without. Regarding the charts, there five Google bar charts, one per phase. These bar charts focus on the difference between the data-stores and this means that the visualization of the results is not normalized in y-axis.

8.4.2 TIME-SERIES POSITIONING PER PLAYER

This query projects the position of a specific football player of Tromso IL according the time. This output, which is presented by table 8.1, can be projected in a helicopter view, which is very useful for a football manager because it shows the game performance in the same way as the manager illustrates the tactics to his players, via the tactics board. Furthermore, this specific view unravels cases of fault player's positioning that a video recording hides. The reason is the fact that a video does not show the whole field during a football game, but focuses only on the part of the field where the ball is. As a result, during the performance evaluation the coaching stuff can focus on only one player in the field.

Table 8.1: A sample of "Time-series positioning per player" query output.

X	Y
32.6357	23.3484
32.6357	23.3484
32.6357	23.3484
32.6357	23.3484
28.7948	23.3488
28.7948	23.3488
28.7948	23.3488
28.7948	23.3488
28.7948	23.3488
28.7948	23.3488

The queries which have been created for this phase are the following:

Table 8.2: MySQL and Mongo queries for the first phase

SELECT x_pos as X, y_pos as Y FROM tromso.tromso_anji WHERE tag_id=1;	db.getCollection('tromso_anji') .find({"tag_id":1}, {x_pos:1, y_pos:1})

As it can be seen by table 8.3, MongoDB has an average execution time 0.0145 seconds. More specifically, MongoDB's slowest execution time was 0.028 seconds and its fastest was 0.01 seconds. On the other hand, MySQL executed the query with average execution time records 0.1375 seconds (excluding fetching time) and 0.274 seconds (including fetching time). Precisely, the highest execution time records of MySQL were 0.215 seconds (excluding fetching time) and 0.432 seconds (including fetching time). In the same fashion, the lowest MySQL's execution time records were 0.251 and 0.14 seconds respectively.

Table 8.3: Execution time of "Time-series positioning per player" query.

Experiment	MongoDB	MySQL	MySQL (without Fetch)
1st	0.028	0.432	0.215
2nd	0.011	0.255	0.124
3rd	0.01	0.251	0.135
4th	0.012	0.275	0.14
5th	0.013	0.245	0.127
6th	0.011	0.261	0.12
7th	0.01	0.264	0.156
8th	0.017	0.251	0.118
9th	0.022	0.253	0.123
10th	0.011	0.233	0.117

As a result, MongoDB was around 9.5 times faster than MySQL without taking into consideration the fetching time and around 19 times faster if the fetching time is included. Moreover as it can be seen by Figure 8.4, the average difference in execution time between MongoDB and MySQL was 0.123 seconds (excluding fetching time) and 0.2595 seconds (including fetching time).

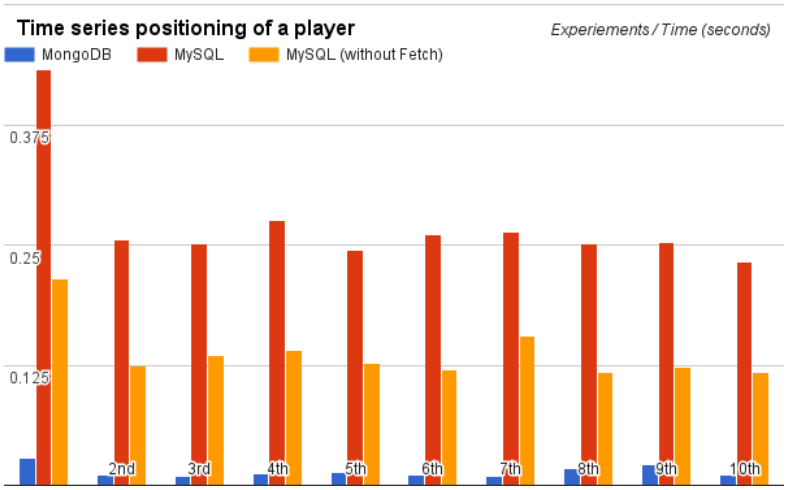


Figure 8.4: Execution time of "Time-series positioning per player" query per data-store.

8.4.3 Times-series positioning per team

This query projects the position of a the whole team of Tromso IL according the time. This output, which is presented by table 8.4, can be projected in a helicopter view and evaluate Tromso IL in terms of tactics. As a result, the team's positioning according the time can prove the football theory, either can indicate the mistakes that Tromso IL commit during the match.

Table 8.4: A sample of "Times-series positioning per team" query output.

Player	X	Y
2	35.2178445647	30.1527231609
6	47.1549917658	19.9495456056
7	41.7908330116	49.8916943603
8	53.2812226712	41.9960580942
10	39.4537328082	33.2552685595
11	44.1981966373	28.8088977197
12	34.3806270711	38.5328679139
13	35.8532058624	22.6150295869
15	42.7558125501	42.2577470748
16	35.3014	47.5268

The queries which have been created for this phase are the following:

Table 8.5: MySQL and Mongo queries for the second phase

SELECT tag_id as Player, x_pos as X, y_pos as Y FROM tromso.tromso_anji WHERE tag_id=1;	db.getCollection('tromso_anji') .find({ }, {tag_id:1, x_pos:1,_pos:1})

As it can be seen by table 8.6, MongoDB has an average execution time 0.002 seconds. More specifically, MongoDB's slowest execution time was 0.004 seconds and its fastest was 0.001 seconds. On the other hand, MySQL executed the query with average execution time records 0.00526 seconds (excluding fetching time) and 1.596 seconds (including fetching time). Precisely, the highest execution time records of MySQL were 0.034 seconds (excluding fetching time) and 1.6913 seconds (including fetching time). In the fashion, the lowest MySQL's execution time records were 0.013 and 1.5283 seconds respectively.

Table 8.6: Execution time of "Time-series positioning per team" query.

Experiment	MongoDB	MySQL	MySQL (without Fetch)
1st	0.003	1.5374	0.034
2nd	0.001	1.5283	0.0013
3rd	0.004	1.6803	0.0023
4th	0.002	1.5832	0.0022
5th	0.001	1.5403	0.0013
6th	0.001	1.5662	0.0022
7th	0.002	1.5768	0.0038
8th	0.002	1.6913	0.0023
9th	0.001	1.6513	0.0013
10th	0.003	1.6049	0.0019

As a result, MongoDB has the same performance as MySQL without taking into consideration the fetching time and around 845 times faster if the fetching time is included. Moreover as it can be seen by Figure 8.5, the average difference in execution time between MongoDB and MySQL was 0.00126 seconds (excluding fetching time) and 1.6909 seconds (including fetching time).

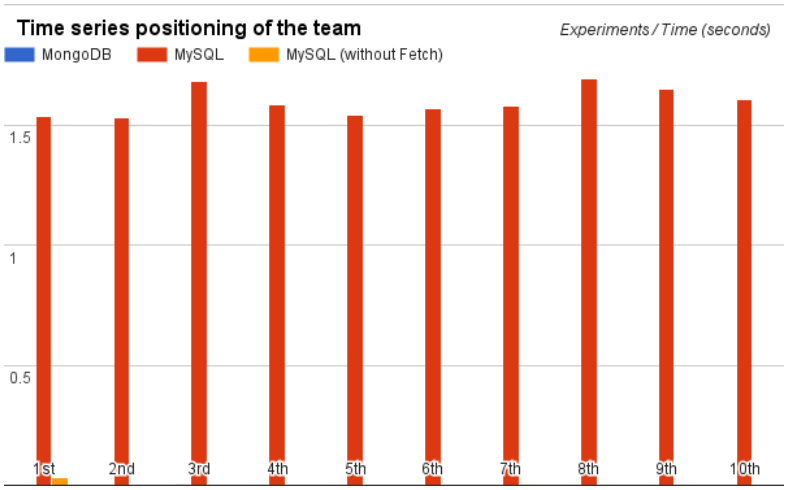


Figure 8.5: Execution time of "Time-series positioning per team" query per data-store.

8.4.4 POSITION OVERLAPPING

Position overlapping occurs in a football match when a player tends to be in a place, where a teammate of him should be. More specifically, this player leaves his position and acts inside the area of his teammates. This event is not welcome, because the player leaves his place unprotected and at the same time he bothers his teammates by reducing their space. In order to achieve this query, it is recommended to apply a JOIN clause.

The SQL Joins clause in a database, combines records from two or more tables of this specific database. More specifically, a JOIN combines fields from two tables by using values common to each table. In our case, we would like to combine the only table, which exists in the database with itself on the "x_pos" and "y_pos" fields. The result will be all the rows from the original and the duplicate table where the join condition is met. In order to retrieve these results, the following query has been formulated:

Table 8.7: SQL JOIN for the third experiment

```
SELECT *
FROM tromso.tromso_anji AS A
JOIN tromso.tromso_anji AS B ON FLOOR(A.x_pos) = FLOOR(B.x_pos)
WHERE FLOOR(A.y_pos) = FLOOR(B.y_pos)
AND A.tag_id <B.tag_id
AND SECOND(A.timestamp) = SECOND(B.timestamp)
AND MINUTE(A.timestamp) = MINUTE(B.timestamp)
AND HOUR(A.timestamp) = HOUR(B.timestamp);
```

However, any version of this JOIN performs poor. More specifically, either the query time extends the limit of 90 minutes and it is aborted either the memory 12 Gb is not enough. As a result, the combination of MySQL workbench with the specific hardware and input cannot produce any results for this demand.

On the other hand MongoDB does not offers any option for JOIN at all. Trying to elaborate why MongoDB cannot support JOIN clause the following explanation is given by the official documentation. The advantages of MongoDB's document data model is the fact that it is flexible and provides developers many options in terms of modeling. The reason is that the most of the time all the data for a record tends to be located in a single document. As a result, for the operational application, accessing data is simple, high performance, and easy to scale with this approach.

However, when it comes to analytics and reporting, it is possible that the data are stored in multiple collections. MongoDB does not support joins. A solution could be the fact that some data is denormalized, or stored with related data in documents to remove the need for joins. Unfortunately, there are cases that this technique cannot be applied.

In general MongoDB applications use two well-known methods for relating documents:

- Manual references where the _id field of one document is saved in another document as a reference. Trying to make a connection with SQL this technique is like the external key use. As a result, inside the main query, a second query can be implemented, which

returns the related data. These references are simple and sufficient for most use cases.

- DBRefs which are references from one document to another using the value of the first document's fields, such as _id field, collection name, and, optionally, its database name. More specifically, with DBRefs allow documents located in multiple collections to be more easily linked with documents from a single collection. To resolve DBRefs, additional queries must be performed in order to return the referenced documents.

To sum up, this phase of the experiment failed.

8.4.5 AVERAGE SPEED PER PLAYER

This query projects the average speed of each football player of Tromso IL. This output, which is projected by table 8.8 consists of a useful metric for effort measurement. Since football has become a show, nowadays, fans get excited for twinkle-toed stars and attractive, eye-catching football. However, coaches claim that there is no substitute for good, old-fashioned effort. As effort, the coaching community defines the football player's dedication to the tactics plan. Football tactics are demanding in terms of positioning and they demand the players to be in constant movement in order to fulfill their obligations. As a result, a good metric of players' effort is the average speed during the game. Fifa claims that a football game is everything but an average running speed. It is more a series of explosive situations [48]. However, a football player who actually works hard during the game and runs constantly has a higher average speed than a fastest but "lazy" teammate.

Table 8.8: A sample of "Average speed per player" query output.

Player	AverageSpeed
1	0
2	1.9937899483034127
3	0
4	0
5	0.10651108360005974
6	2.362199170637215
7	1.8580462111975549
8	2.215603094642742
9	0.0661225283958499
10	2.281123177099688

The queries which have been created for this phase are the following:

Table 8.9: MySQL and Mongo queries for the fourth phase

SELECT tag_id as Player, AVG(speed) as AverageSpeed FROM tromso.tromso_anji GROUP BY tag_id	db.getCollection('tromso_anji') .aggregate([{$group: {_id:'$tag_id', AverageSpeed: {$avg:'$speed'} } }])

As it can be seen by table 8.10, MongoDB has an average execution time 0.6878 seconds. More specifically, MongoDB's slowest execution time was 0.794 seconds and its fastest was 0.66 seconds. On the other hand, MySQL executed the query with average execution time records 0.4753 seconds (for this query, fetching time was close to 0.0001 second and, as a result, it is not included, since it is not affecting the results). Precisely, the highest execution time records of MySQL was 0.58 seconds and the lowest was 0.422 seconds.

Table 8.10: Execution time of "Average speed per player" query.

Experiment	MongoDB	MySQL
1st	0.794	0.462
2nd	0.699	0.58
3rd	0.672	0.484
4th	0.66	0.523
5th	0.667	0.502
6th	0.669	0.435
7th	0.678	0.422
8th	0.664	0.44
9th	0.678	0.455
10th	0.697	0.45

As a result, MySQL was around 1.5 times faster than MongoDB. Moreover as it can be seen by Figure 8.6, the average difference in execution time between MongoDB and MySQL was 0.2125 seconds.

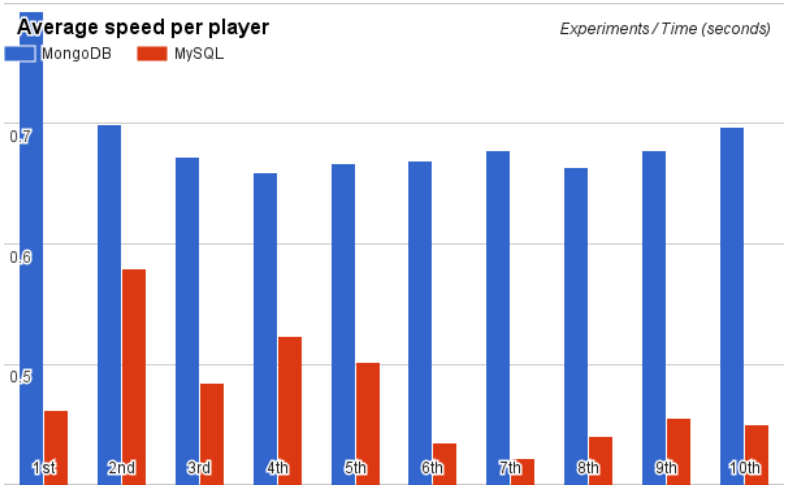


Figure 8.6: Execution time of "Average speed per player" query per data-store.

8.4.6 TEAM FORMATION BY PLAYER'S MEDIAN POSITIONING

This query projects the median position of each player of Tromso IL. This output, which is presented by table 8.11, can be projected in a helicopter view, which can show which is the natural position the players. More specifically, this view can indicate the center of the area, where a player moves during the match and after comparing this area with the tactics, a coach can conclude if this player matches to his current or to another position inside teams formation. Moreover, this team's view either can prove team's formation, either can indicate a new formation for this specific team.

Table 8.11: A sample of "Team formation by player's median positioning" query output.

Player	MedianX	MedianY
1	31.591419600300448	20.84892256026343
2	49.573679231745764	36.15064125849095
3	71.05233086857395	71.85746803105224
4	69.60789999999999	70.26800000000001
5	44.30664857490613	66.81009417634121
6	46.81089716353267	33.56421631533662
7	53.560309413904186	45.25000258615213
8	48.9803184664832	38.33945179307242
9	44.77807486006899	65.34691900267752
10	49.60221696827333	36.704427571323

The queries which have been created for this phase are the following:

Table 8.12: MySQL and Mongo queries for the fifth phase

SELECT tag_id as Player, AVG(x_pos) as MedianX, avg(y_pos) as MedianY FROM tromso.tromso_anji GROUP BY tag_id	db.getCollection('tromso_anji') .aggregate([{ $group: {_id:'$tag_id', MedianX: { $avg:'$x_pos'} , MedianY: { $avg: '$y_pos'} }])

As it can be seen by table 8.13, MongoDB has an average execution time 0.7329 seconds. More specifically, MongoDB's slowest execution time was 0.783 seconds and its fastest was 0.71 seconds. On the other hand, MySQL executed the query with average execution time records 0.4896 seconds (for this query, fetching time was close to 0.0001 second and, as a result, it is not included, since it is not affecting the results). Precisely, the highest execution time records of MySQL was 0.582 seconds and the lowest was 0.427 seconds.

As a result, MySQL was again around 1.5 times faster than MongoDB. Moreover as it can be seen by Figure 8.7, the average difference in execution time between MongoDB and MySQL was 0.2433 seconds.

Table 8.13: Execution time of "Team formation by player's median positioning" query.

Experiment	MongoDB	MySQL
1st	0.71	0.478
2nd	0.724	0.487
3rd	0.745	0.427
4th	0.733	0.491
5th	0.715	0.468
6th	0.717	0.45
7th	0.74	0.563
8th	0.742	0.471
9th	0.72	0.582
10th	0.783	0.479

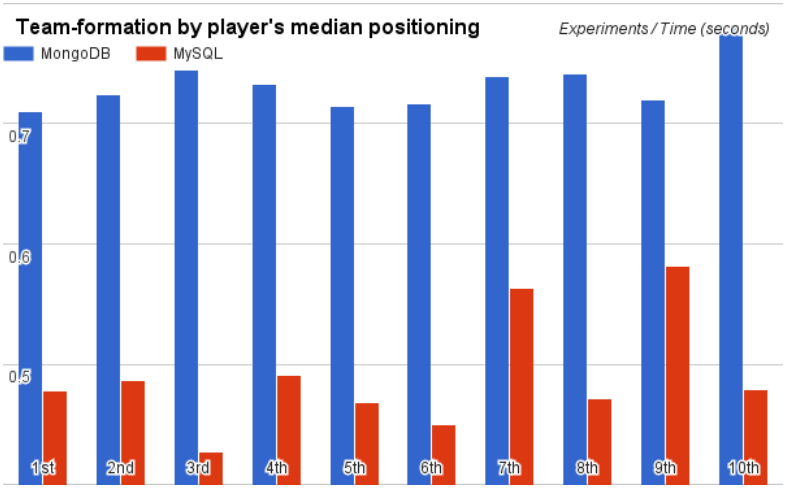


Figure 8.7: Execution time of "Team formation by player's median positioning" query per data-store.

9 DISCUSSION

In this section a discussion takes place regarding the comparison between SQL and NoSQL data stores. More specifically, this discussion is based to the previous sections of this document and it is split into two parts. The former refers to the theoretical part, which is placed in Chapter 4, Chapter 5, Chapter 6 and Chapter7. The latter focuses on the experimental part, which is described in Chapter 8. After these two parts, there is a conclusion with the finals thoughts regarding this work.

9.1 SQL VS NOSQL

By inspecting the theoretical part of this work, some interesting conclusions can be made:

NoSQL cannot supersede SQL A good parallelism would be like claiming that boats were superseded by cars because of the fact that cars consist of a newer technology. Of course SQL and NoSQL do the same thing: both of them store data. However, they apply different approaches, which are more or less suitable than the other for a project. As a result, the nature of the project will introduce if NoSQL is a replacement for SQL for this specific project. Despite of that, in general, the feeling newer technology does not equal that NoSQL a replacement for SQL. The proper term would be an alternative.

NoSQL is not better or worse than SQL Trying to extend the first conclusion, a remark must be mentioned: not all the project are the same, in terms of demands. For some projects, a SQL database is more suitable and for some others, the requirements are better suited to a NoSQL database.

Between SQL and NoSQL, there is not a clear distinction Any claim regarding clear differences between these two data stores is not necessarily true. More specifically, a lot NoSQL databases are adopting SQL features and vice versa. In many cases, according the time the choices between SQL and NoSQL are likely to become increasingly darken. In addition, in the near future some NewSQL hybrid databases could provide some brand new features which will make the distinction between SQL and NoSQL to difficult.

As a result, there is no clear answer to the question "SQL or NoSQL", because each data store has advantages, disadvantages and similarities with the rest. Whatever holds for the programming languages can be assumed also for the database. Each database and technology should be evaluated according to a specific project and specific demands.

9.2 MYSQL VS MONGODB

In contrast with the previous section, by observing the results of the experimental part, some more technical conclusions can be made about the "SQL vs NoSQL", regarding the needs of the "Soccer Video and Player Position Data-set" project, which is introduced in Chapter 3. The conclusions are based on the comparison between MySQL and MongoDB - the two most popular SQL and NoSQL databases in 2016 and are organized in the following way:

Tables vs Documents MySQL provides a store of related data tables. More specifically, every row is a different record, with strict structure and field types. On the other hand, MongoDB stores data in JSON-like field-value pair documents, which can be stored in collections. The main difference between MongoDB and MySQL is the fact that in the former the user do not have restrictions regarding the field types. MySQL tables have a strict data template. As a result they are demanding in data-types and at the same time they are not prone to mistakes. Practically, this means that MySQL database validates the input data in terms of data-types, which can cause slow performance when a project requires just tank, which can store data that will be evaluated in the application level. On the opposite side, MongoDB is more "flexible" and "forgiving", but this can lead to consistency issues.

Schema vs Schemaless In MySQL database, before any action a schema is required. This means that before input any data, the database should have all required the tables and their field data-types. In addition, the schema should involve other information like:

primary keys - values, which uniquely identifies each record in a database table.

indexes - data structures, which improve the speed of data retrieval operations

relationships - logical connections between data fields

To sum up, MySQL schema must be produced before any data input and manipulation. In addition, it is difficult to change, in the future except some updated which might destroy the normalization of the database form.

On the other hand, in a MongoDB database, data can be added freely anytime and anywhere. More specifically, there are no restrictions for the user to specify any document design or even a collection up-front. Consequently, MongoDB database may be more suited to projects where the initial data requirements are unknown or difficult to be specified.

Performance The most interesting part of this discussion and the most difficult to be defined is the performance race of these database systems. In other words, the question which system is fastest cannot be simply answered. Theory claims that NoSQL is faster than SQL in terms of querying. NoSQL's (MongoDB in this case) simpler denormalized store allows user to retrieve all the information about a specific record in a single request. This is also proven by the fact that in phase one and two of experimental part, MongoDB was from 10 to 845 times faster than MySQL. On the other hand, for more complex queries, which require calculation, such as the phase four and five of the experimental part, MySQL performed 1.5 times better than MongoDB. Something that must be mentioned again is that, always, the project design and data requirements have the most impact on performance. More specifically, a well-designed MySQL database performs better than a poorly designed MongoDB equivalent and vice versa.

Relational JOIN A very popular aspect of MySQL is the powerful JOIN clause. More specifically, JOIN clause offers to the user the ability to obtain data from multiple tables using

just a single SQL statement. However, MongoDB has no equivalent of JOIN, and this is a huge disadvantage for those for who want to switch from SQL to NoSQL. A solution for this problem is to normalize the collections and manually link the relevant records between different collections. As a result, for project where the information retrieval across different relations are essential, then SQL is the only way. In practice, the third phase of experimental part failed, because Mongo couldn't implement a JOIN and MySQL JOIN failed.

SQL vs NoSQL Scaling Dealing with big data requires systems, which can adapt as data grows. A typical example is the moment when it is necessary to distribute the load of the data among multiple servers. This is not trivial for SQL-based systems since the whole database is stored in one server. Of course there are solution such as clustering: multiple servers have access to the same central store. On the other hand, NoSQL's simpler data models can achieve an easier process of the data, and many, such as MongoDB, have been built with scaling functionality. As a result, for project which require horizontal scaling, NoSQL seems to be ideal.

9.3 SUMMARY

To sum up, NoSQL and SQL databases treat the same needs in different ways. As software evolution indicates, the requirements change according time, and thus it is possible one database which seems ideal for a specific application to become an issue in the near future. All this work conclude to the following proposals:

SQL fits to projects :

- with logical related discrete data requirements which can be identified up-front
- when data integrity is essential
- when standards-based proven technology with good developer experience and support is needed

NoSQL fits to projects :

- with unrelated, indeterminate or evolving data requirements
- with simpler or looser project objectives, able to start coding immediately
- when speed and scalability is imperative.

In the case of input data 3, a NoSQL database appears the most practical option if the main intention of the database is to store data which will be retrieved by an application for a visual project. On the other hand, if the data are stored in order to apply analytics or business intelligence then SQL database is the solution.

10 FUTURE WORK

Since NoSQL and Big Data, are newborn fields of our science there are a lot of parameters, which could be improved in this work, in order to offer a better overview of this SQL vs NoSQL comparison

10.1 NOSQL DBMS

In the same framework more NoSQL databases can be examined. According to http:db-engines.com:, the NoSQL of the top-30 DBMS, which are not included in this work are the following:

- Hadoop/Hbase
- Hypertable
- RavenDB
- Redis
- Amazon DynamoDB
- CouchBase

10.2 INPUT

Despite the football analytics, NoSQL databases can be used in every possible project which requires data storage. However, there are two major categories, in which data storage requirements are split and according to them the the project are divided into

Transactional (On-line Transaction Processing - OLTP) , which are described by a large amount of short on-line transactions. The main purpose for an On-line Transaction Processing project is achieve very fast query processing, data integrity in multi-access environments and effectiveness.

Analytical (On-line Analytical Processing - OLAP) , which are outlined by low number of transactions. In On-line Analytical Processing data store, queries are complex and involve aggregations. Moreover, as in On-line Transaction Processing case, the response time is an effectiveness measure.

This work investigates the advantages of NoSQL data stores as OLTP database systems. It would be an interesting project for the future, a research regarding the capability of NoSQL to fulfill OLAP requirements.

REFERENCES

[1] "Benchmarking Cassandra Scalability on AWS - Over a million writes per second". Retrieved January 27, 2015.

[2] "The importance of user-friendly technology" By Dave Kearns. Network World. Retrieved January 27, 2015.

[3] "An Introduction To NoSQL & Apache Cassandra". Retrieved January 27, 2015.

[4] "Soccer video and player position data-set": S. A. Pettersen, D. Johansen, H. Johansen, V. Berg-Johansen, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, H. K. Stensland, and P. Halvorsen, in Proceedings of the International Conference on Multimedia Systems (MMSys), Singapore, March 2014. Retrieved January 27, 2015.

[5] "Introduction to Document Databases with MongoDB". This article originally appeared in the June 2013 issue of Web & PHP. Retrieved May 21, 2015.

[6] Exploring the Different Types of NoSQL Databases Part II. Retrieved June 1, 2015.

[7] "NoSQL distilled": A brief guide to the emerging world of polyglot persistence. Pramod J.Salalage, Martin Fowler.

[8] "Introduction to MongoDB".Official Documentation of MongoDB. Retrieved May 21, 2015.

[9] "MongoDB System Properties".DB-Engines. Knowledge Base of Relational and NoSQL Database Management System. Retrieved May 26, 2015.

[10] "Introduction to MongoDB Query Language". Safari Professional Books. Retrieved May 21, 2015.

[11] "Do Things Big". Official Documentation of MongoDB. Retrieved May 26, 2015.

[12] MongoDB Source Code. Official Documentation of MongoDB. Retrieved May 26, 2015.

[13] "3,000+ messages each month on the user forum".Retrieved May 26, 2015.

[14] "Public Production Deployments".Retrieved May 26, 2015.

[15] Lots of job postings for MongoDB devs.Retrieved May 26, 2015.

[16] 100,000+ downloads/month for MongoDB. Retrieved May 26, 2015.

[17] "User groups and events There were 500+ people at MongoSV, we expect a similar turnout for MongoSF, and there are user groups in many cities. e.g. the NY MUG has over 700 members". Retrieved May 26, 2015.

[18] Meet Azure DocumentDB. Official webpage of Azure DocumentDB. Retrieved May 27, 2015.

[19] Meet Azure DocumentDB. "Introducing DocumentDB" by David Chappell. Retrieved June 2, 2015.

[20] Manage DocumentDB capacity needs. Retrieved June 2, 2015.

[21] "Introduction to Apache Cassandra". White papper by Datastax Corporation. Retrieved June 2, 2015.

[22] Official Apache Cassandra web-page. Retrieved June 10, 2015.

[23] "Why does Scalability matter, and how does Cassandra scale?". Retrieved June 10, 2015.

[24] DB-Engines Ranking. The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly. Retrieved June 2, 2015.

[25] "Real-time business intelligence is going mainstream, thanks in part to the Storm and Spark open source projects. Here's how to choose between them". Retrieved June 11, 2015.

[26] "Real-time business intelligence is going mainstream, thanks in part to the Storm and Spark open source projects. Here's how to choose between them". Retrieved June 11, 2015.

[27] Apache Storm official web-page. Retrieved June 11, 2015.

[28] Apache Storm vs. Apache Spark. Retrieved June 11, 2015.

[29] Introduction to Apache Storm on HDInsight: Real-time analytics for Hadoop.. Retrieved June 11, 2015.

[30] Introduction to Apache Storm. Retrieved June 11, 2015.

[31] "Basic Storm Concepts". Retrieved June 11, 2015.

[32] SQL for Apache Storm. Retrieved June 11, 2015.

[33] Chapter 1. "Introduction to Data Analysis with Spark". Retrieved June 11, 2015.

[34] Introduction to Apache Spark with Examples and Use Cases. Retrieved June 11, 2015.

[35] Big Data Processing with Apache Spark âĂŞ Part 1: Introduction. Retrieved June 11, 2015.

[36] Spark Community. Retrieved June 11, 2015. https://spark.apache.org/community.html

[37] Travelling salesman problem. Retrieved November 4, 2015.

[38] "DB-engines/Neo4j". The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly. Retrieved November 4, 2015.

[39] Neo4j Graph Database Official web-site. Retrieved November 4, 2015.

[40] "Cypher". About Cypher. Retrieved November 21, 2015.

[41] "Neo4j News: Introducing Neo4j 2.2". Retrieved November 21, 2015.

[42] " How to connect to Neo4j?". Retrieved November 21, 2015.

[43] "Understanding Neo4j Scalability". David Montag. January 2013. Retrieved November 21, 2015.

[44] "OrientDB official webpage". Retrieved December 7, 2015.

[45] "Why OrientDB?". Retrieved November 21, 2015.

[46] "API OrientDB". Retrieved November 21, 2015.

[47] "Magic Quadrant for Operational Database Management Systems". Retrieved November 21, 2015.

[48] "Physical Analysis of the FIFA Women's World Cup Germany 2011". Retrieved November 21, 2015.