



Trabajo de Final de Grado

GRADO DE INGENIERÍA INFORMÁTICA

**Facultad de Matemáticas e Informática
Universidad De Barcelona**

ShareCare: a study of databases within Q&A webapp context

Pablo Almécija Rodríguez

Director: Guillermo Blasco Jiménez
Realizado en: Departamento de Matemáticas e Informática

Abstract

The content of this work is related to the study of the different database families that can be found working in different applications for the web. This study means a description of those families together with an analysis of their features and some of their common uses.

This overview of databases will be reinforced by the example of a web application, created to exemplify some interesting use cases that make nowadays application use several kinds of databases.

The purpose of this work is, therefore, to show how important is to know the possibilities that exist in terms of databases, and also to know some facts that a developer may bear in mind in order to make a good choice when selecting a database to work with.

Table of contents

1. Introduction	5
2. Application	7
2.1. How it works	7
2.1.1. Prerequisites	7
2.1.2. Starting up	7
2.1.2.1. Secure password creation	9
2.1.3. Home	9
2.1.4. Profile	11
2.1.5. Ask question	11
2.1.6. Questions	12
2.2. Use cases	14
3. Databases	18
3.1. Taxonomy of DBMS	18
3.1.1. General-purpose DBMS vs Special-purpose DBMS	20
3.1.2. A better way to classify DBMS – Data model	21
3.1.2.1. RDBMS	21
3.1.2.2. KV Store	24
3.1.2.3. Document store	26
3.1.2.4. Graph Databases	29
3.1.2.5. Column-wide Databases	32
3.2. Databases information tables	35
4. DBMS in <i>ShareCare</i>	39
4.1. Storing user information	40
4.1.1. Using MariaDB	40
4.2. Liking or disliking questions	42
4.2.1. Using Redis	42

4.3. Uploading documents to complete their posts	43
4.3.1. Using AWS S3	44
4.4. Making full text searches	44
4.4.1. Using ElasticSearch	45
5. Deployment	49
5.1. Preparing the environment	49
5.2. Starting <i>ShareCare</i> for the first time	52
5.2.1. Creating JAR	52
5.2.2. Creating data structure for MariaDB and ElasticSearch	53
6. Technologies involved	54
6.1. Integrated Development Environment – IntelliJ IDEA	54
6.2. Spring Framework	54
6.3. Maven	56
6.4. Integrating databases	56
6.4.1. ElasticSearch	56
6.4.2. Redis	57
6.4.3. MariaDB	57
6.4.4. Amazon Web Services – Simple Storage Service (AWS S3)	58
7. Conclusion	59
8. References	61

1. Introduction

This work is about the development of a web application that involves the use of different kinds of database management systems in order to accomplish different tasks in an effective way. Along the work, an analysis of database families will take place, studying their structure and main features, as well as the analysis of the specific databases that are used in the web application *ShareCare*, explaining why they are good for a task within it.

Nowadays, finding a web application with no database is almost inconceivable. Whenever we look for information on any topic and enter a forum, we are quickly asked if we want to join the community, to *create an account*. Most of us have email accounts, either for personal or work purpose, perhaps Facebook, which knows our friends, family, what we like or what we do. Or just a computer that is connected to the Internet, and every now and then, our operating system may ask us if we would be so kind as to give them the information of, let's say, the performance of our computer, so they can enhance your user experience. All those scenarios mean obtaining data and storing it, whatever the reason may be.

No matter what kind of software we may think of, as far as it deals with some information, such as item inventories in a store or clients information in a bank, there is going to be a DBMS behind the scenes managing the data. But there are just so many kinds of databases, and some business may choose one over another for a number of reasons, may them be how they are going to operate with their data, the cost of the investment, the complexity of the integration, etc. And that decision is quite important, since in this Era of Information, being able to rely on your data the way you need to is essential. And not all the databases do the same jobs as well as any other.

The application itself is called *ShareCare*, and it is a question and answer (from now on, Q&A) web application designed for health professionals to share their doubts and opinions about the diagnose of their patients. This software is supposed to be an application similar to stackoverflow.com, but for private use to medical staff.

Chapter 2, [Application](#), is going to be a walk through the application, showing the different parts of the software from the point of view of the user. There, all the different views and actions a user can perform in the application in its current state.

Chapter 3, [Databases](#), contains the description, uses and analysis of the main different families of databases that exist nowadays, and giving some examples in which they are widely used.

The analysis carried out in the previous chapter will be then applied to the specific case of the software *ShareCare* in chapter 4, [DBMS in ShareCare](#). There, the different use cases within *ShareCare* that are related to the use of databases will be explained, showing

what capabilities of what database family are more relevant to perform a task, and therefore, proving the reason why it was selected.

The instructions on how to deploy this project on a server will be in chapter 5, [Deployment](#). Those will be all the necessary steps to take in order to have the application up and running, ready to be used by its users.

In chapter 6, [Technologies involved](#), we will proceed to an explanation of the different technologies used to develop the application.

Last, but not least, at the end of this document there will be two final chapters devoted to a [Conclusion](#), which will sum up the main ideas learnt through this work, and [References](#).

2. Application

ShareCare is intended to be a private platform where certified professionals who work on the health field can share their experiences with each other, ask doubts and help their colleagues find the best treatment for their patients.

This software has been implemented as a Q&A web application, with the structure of a discussion forum. Any user allowed in the system can collaborate in the questions that are posted, or can ask new questions.

As the application has been implemented with academic purposes, its functionalities are limited and it is obviously not ready for a real environment.

Along this section, we will see how a user is supposed to use the application.

2.1. How it works

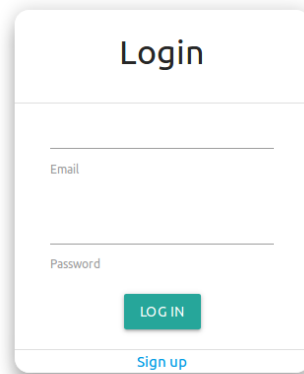
2.1.1. Prerequisites

As mentioned before, the application is intended for the exclusive use of health professionals. The ideal situation would be that health institutions, may them be hospitals, health centers, etc., would send their doctors an invitation to join the platform. This implies a high level of collaboration from the government.

Once a user has received the invitation, he could be able to register into the system and start using it.

2.1.2. Starting up

Once the user has been invited to the platform, the first thing he needs to do is create his account. When entering the URL in the browser (again, so far there is no real URL, since there is no domain for the application), the user will face the login screen:



A login form titled "Login" with a white background and rounded corners. It features two input fields: "Email" and "Password". Below the "Password" field is a green "LOG IN" button. At the bottom of the form is a blue "Sign up" link.

Figure 1: Login and sign up screen

Figure 1 shows two possible courses of action:

1. If the user has already signed up, he can now log in using his email and password.
2. If the user has not signed up yet for the system, then he needs to do it.

Register

Figure 2: Register view

In case number 2, the user must sign up. Clicking in the link to sign up takes the user the view shown in figure 2. There, the user will need to enter his name, email and password to become part of the system.

2.1.2.1. Secure password creation

During this first step to get into the web application, if the user is not registered yet, he has to sign up. This is a process where the user will enter some data, and pick a password. Although web security is not the focus of this work, it is worth mentioning that a safe login/register process has been implemented.

This safe process means that the password chosen by the user is not stored in the database. Instead, there is a technique called “Hash&Salt”. Using the password enter by the user, these two keys are created. Salt is going to be a sequence of 32 random characters, which will be store in the database. Then, a hash function will be applied to the concatenation of the salt and the password, resulting into a value that will be stored too.

Using this method, we will not have the password stored in the database, which would not be legal and it would be easily hackable.

Now, when the user wants to log in, the salt attribute will be retrieved from the database, and together with the password provided to log in, the same hash function that was perform to create the hash value when signing in will happen. This value will then be compared to the hash value stored in the database, and they should be the same. If they are not, they log in process fails because the password will not be correct. Of course, if the email is not in the database, the process will also fail.

2.1.3. Home

After a successful login, the user will find himself in the homepage, for the time being looking like this:

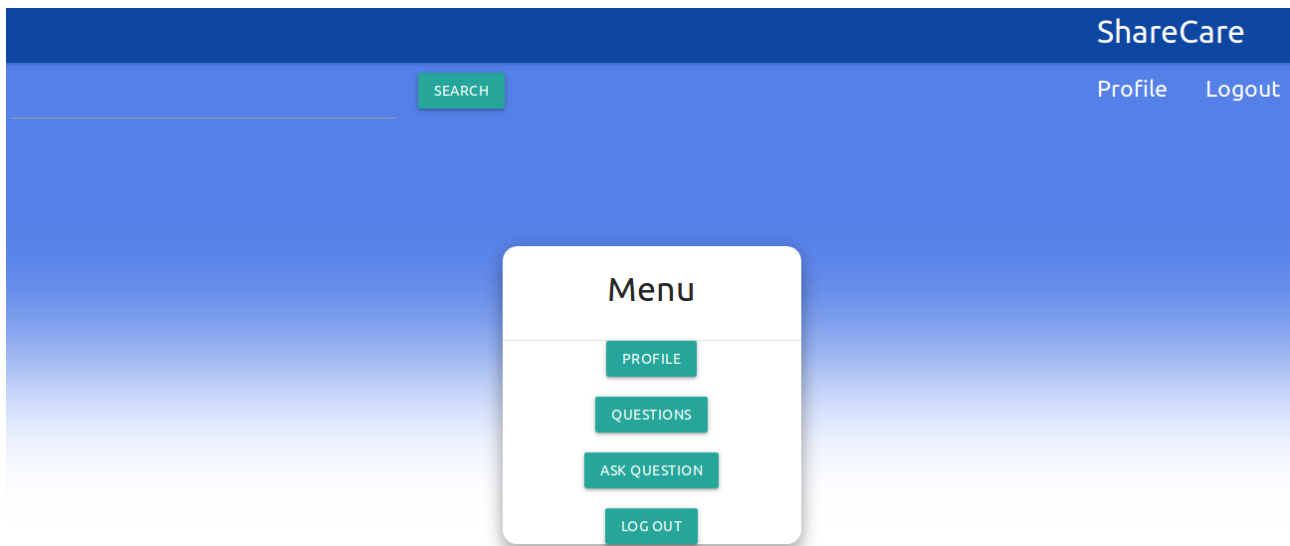


Figure 3: Home page

Once the user is logged in, Spring security will keep the session open for a limited amount of time. This will acknowledge the user an authenticated entity, giving him access to the different views. If there were no authentication service, any user could just write in the address bar the URL of a view, and access it.

After the session expires or the user logs out, this user will not be able to access any view and should log in again.

From here, the user can do everything there is to do in the web app so far:

- In the uppermost left corner, there is a search field. The user can enter any text to look for within the questions that are in the database. This will be seen in the section [Making full text searches](#).
- In the uppermost right corner, there are three clickable elements:
 - *ShareCare* logo, which will bring the user to this view.
 - Profile, that will take the user to his profile in order to change his data.
 - Logout.
- In the center of the view, there is a box with different options, two of which were already in the navigation bar (upper part). The other two are:
 - Questions, which will take the user to a view where he will find all the questions in the database.
 - Ask questions, to open a view where the user can post a question.

2.1.4. Profile

The profile view is where the user can access its personal data and modify them. So far, the user can only change his name. As mentioned above, this screen is accessible from two different places:

1. Clicking on the “Profile” button in Home.
2. From any view in the web app, clicking on profile in the navigation bar.

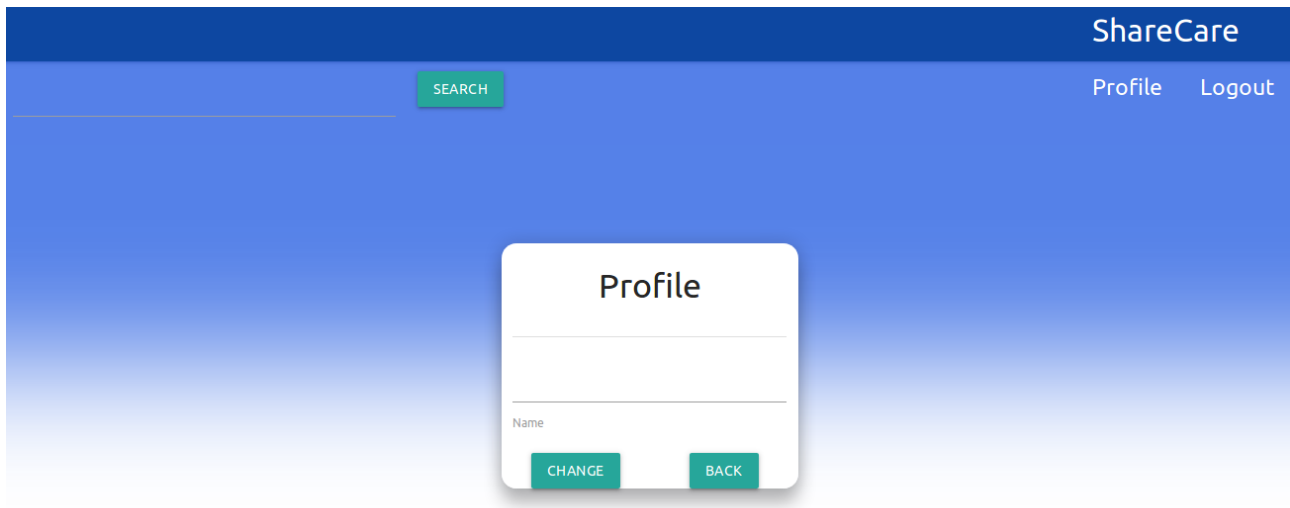


Figure 4: Profile view

For the time being, the only attribute the user can change is his name. Clicking the button change will perform the modifications the user has made to his profile.

2.1.5. Ask question

When asking a question, user must add a title and the body of the message. Thanks to the **WYSIWYG** (What You See Is What You Get) editor, a user can format the body of the message, using bold letters to emphasize a word, using lists, etc.

Also, it is possible to attach a document to the question. So far it is just possible to attach one document. More about how documents are stored in the section [Uploading documents to complete their posts](#).

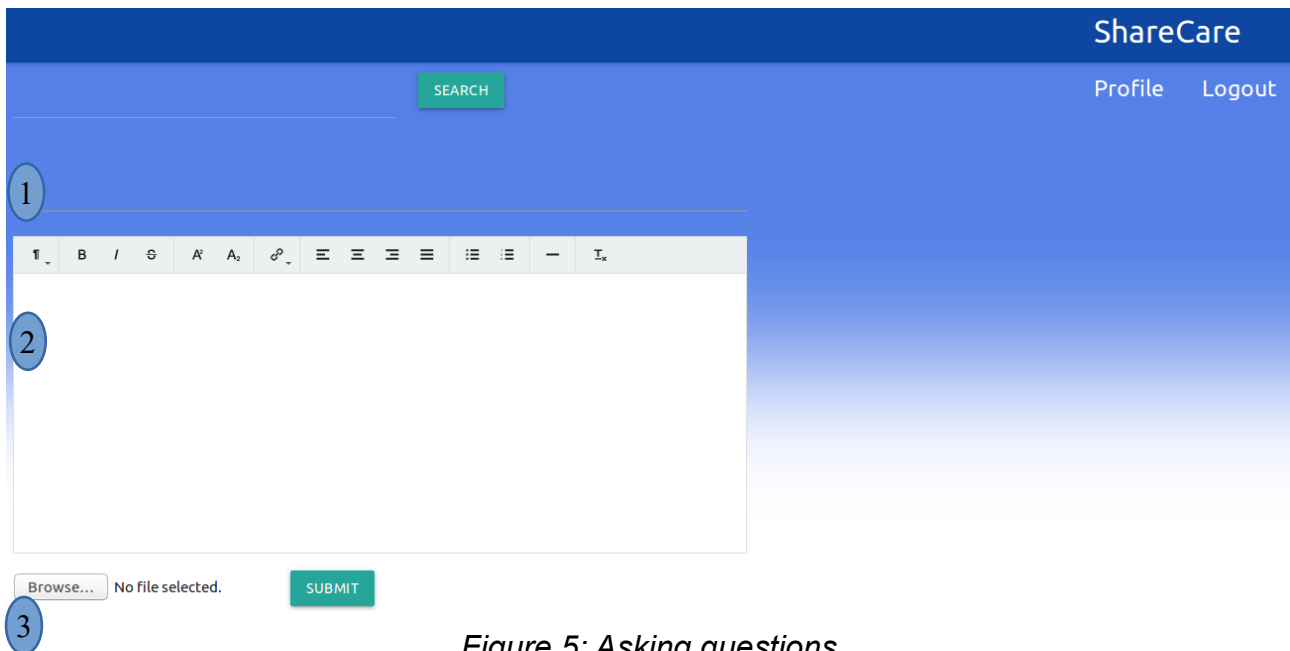


Figure 5: Asking questions

The important items on this view are:

- 1 Title field
- 2 WYSIWYG editor
- 3 Upload file button

2.1.6. Questions

In this view, the user will see all the questions that have been posted. Again, for the purpose of this work, the application does not count with every detail it should to be in the market, it just has enough as to test it and show the different databases used in it.

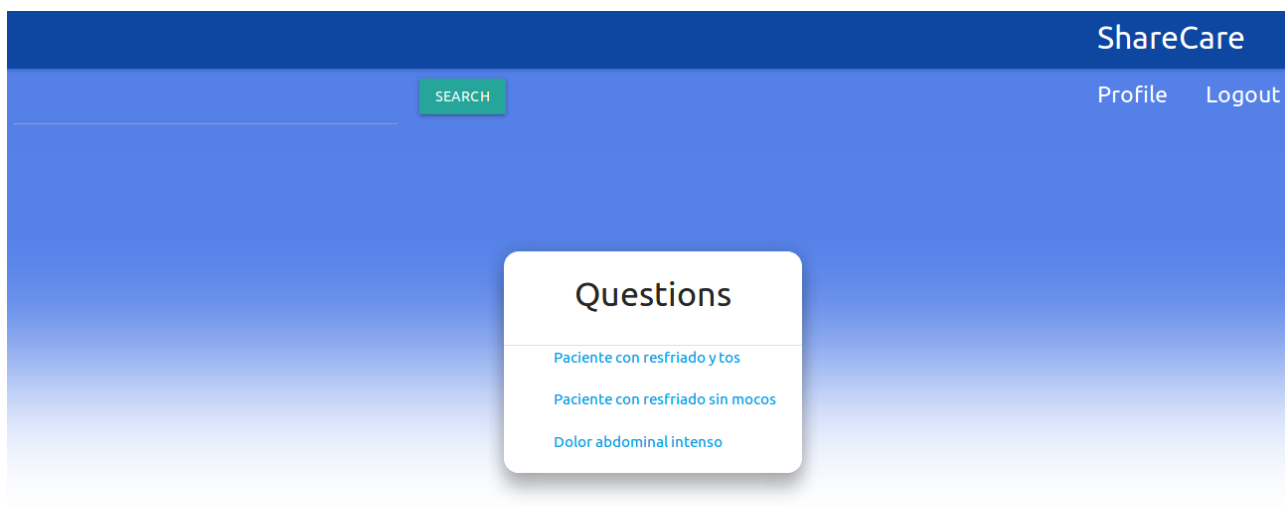


Figure 6: Questions view

The user can now click on any of the questions, and will be taken to that question's view. Within the individual view of a question, users can answer.

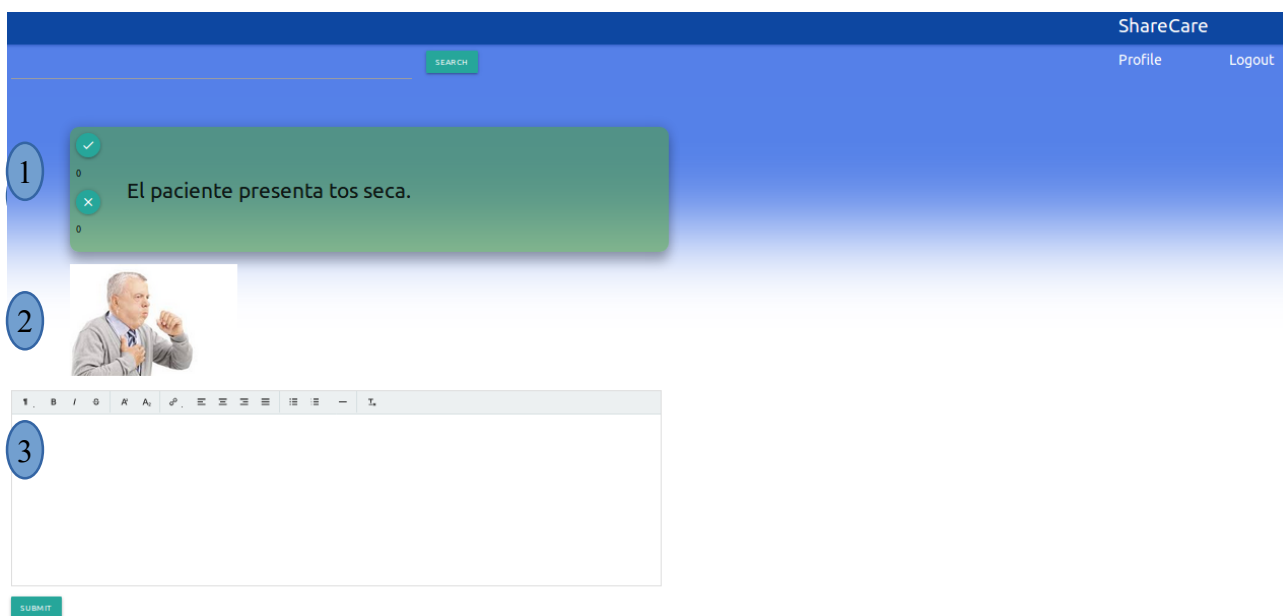


Figure 7: View for a question

In this view, we can appreciate the following elements:

- 1 Question to be answered. The two buttons on the left of the question are there to rate the question. This will be studied in the section [Liking or disliking questions](#).

2 If a document was attach to the question, it will appear here. If it is an image, it will appear as in the example. If it is a document, a link will allow a user to open the file in a new tab.

3 Again, the WYSIWYG editor to answer the question.

2.2. Use cases

In this section we are going to study the current use cases of *ShareCare* in the form of diagrams. Out of those use cases, there are some that will be related to using DBMS, and that part will be dealt with in the section *DBMS in ShareCare*. These use cases will be studied from the point of view of the user.

So far, users of this application will be able to perform the following tasks:

- Register in the platform
- Log in
- Modify their profile
- Ask questions
- See questions
- Upload a document
- Answer questions
- Like or dislike questions
- Make a full text search

Every use case will describe the set of actions necessary in order to carry out each of the tasks.

Use Case 1: Register in the platform		
Precondition	User must have received the invitation to join the platform.	
Description	User will join <i>ShareCare</i>	
Regular flow	Step	Action
		1 User enters <i>ShareCare</i> URL.
		2 User clicks on the link saying "Sign up"
		3 User enters his name, email and sets a password
		4 User submits form
Post condition	User is now register in <i>ShareCare</i> and can log in	
Exceptions	Step	Action
		User did not receive the invitation to join.
	3	User entered an email that is already in the database.
Alternative flow	Step	Action
		Repeat action 3

Table 1. Use Case 1: Register in the platform

Use Case 2: Log in		
Precondition	User must have completed the registration process	
Description	User enter <i>ShareCare</i> web application	
Regular flow	Step	Action
		1 User enters <i>ShareCare</i> URL.
		2 User enters his email and password
		3 User clicks the button saying "Log in"
Post condition	User finds himself in the homepage of <i>ShareCare</i>	
Exceptions	User entered a wrong email and/or password	
Alternative flow	Step	Action
		Repeat action 2

Table 2. Use Case 2: Log in

Use Case 3: Modify profile		
Precondition	User must be logged in.	
Description	User modifies his personal data	
Regular flow	Step	Action
		1 User clicks the button saying "Profile"
		2 User re-writes the fields he wants to modify
		3 User submits form
Post condition	User finds himself in the homepage of <i>ShareCare</i> ; his data changed.	
Alternative flow	Step	Action

Table 3. Use Case 3: Modify profile

Use Case 4: See questions		
Precondition	User must be logged in.	
Description	User will get a view of all questions in ShareCare.	
Regular flow	Step	Action
	1	User clicks the button that says "Questions"
Post condition	User visualizes the questions in the system	
Exceptions	Step	Action
		There are no questions in the system

Table 4. Use Case 4: See questions

Use Case 5: Ask question		
Precondition	User must be logged in.	
Description	User will post a question in ShareCare	
Regular flow	Step	Action
	1	User clicks the button that says "Ask question"
	2	User writes a title for the question.
	3	User writes the body of the question.
	4	[OPTIONAL] Use case 6 .
	5	User submits form.
Post condition	User is back to homepage; question posted.	
Exceptions	Step	Action
		User did not enter a title or body for the question.
Alternative flow	Step	Action
		Repeat from action 2 or 3 depending on what is missing

Table 5. Use Case 5: Ask question

Use Case 6: Upload picture		
Precondition	User is currently asking a question.	
Description	User will attach a file to his question.	
Regular flow	Step	Action
	1	User clicks the button that says "Browse"
	2	User picks a valid document from the file manager.
	3	User presses "Accept"
Post condition	Document is ready to be uploaded if question is submitted	

Table 6. Use Case 6: Upload picture

Use Case 7: Answer question		
Precondition	User must be logged in and completed use case 4.	
Description	User will post an answer to a question in <i>ShareCare</i> .	
Regular flow	Step	Action
	1	User writes the answer to a question.
	2	User submits form.
Post condition	User stays in the same view; the answer now appear under the question.	

Table 7. Use Case 7: Answer question

Use Case 8: Like or dislike question		
Precondition	User must be logged in, and be within a question view.	
Description	User will rate a question.	
Regular flow	Step	Action
	1	User clicks the button with the check or with the "X", in order to like or dislike the question, respectively
Post condition	User is in the same page; the counter of likes or dislikes is modified.	

Table 8. Use Case 8: Like or dislike question

Use Case 9: Make full text search		
Precondition	User must be logged in.	
Description	User will get a list of questions matching his search.	
Regular flow	Step	Action
	1	User clicks on the search field on the navigation bar.
	2	User writes his query.
	3	User presses "Search"
Post condition	User now visualizes a list of the questions matching his query.	

Table 9. Use Case 9: Make full text search

As we can see, some of these use cases are related to performing different operations on data. The next chapter will deal with different kind of databases, some of which will be used to solve these problems.

3. Databases

The world is full of data that we want to register, store, and work with. Once we have a collection of **organized** data, we called it a **database**. These databases need to be designed thoroughly in order to be useful. That means that the data should model aspects of reality in a way that the information can be processed. A database is a concept that existed long before computers were even invented. For instance, a book keeping the record of the ships entering and departing a certain port in the XVI century is a database, and that has obviously nothing to do with computers at that time.

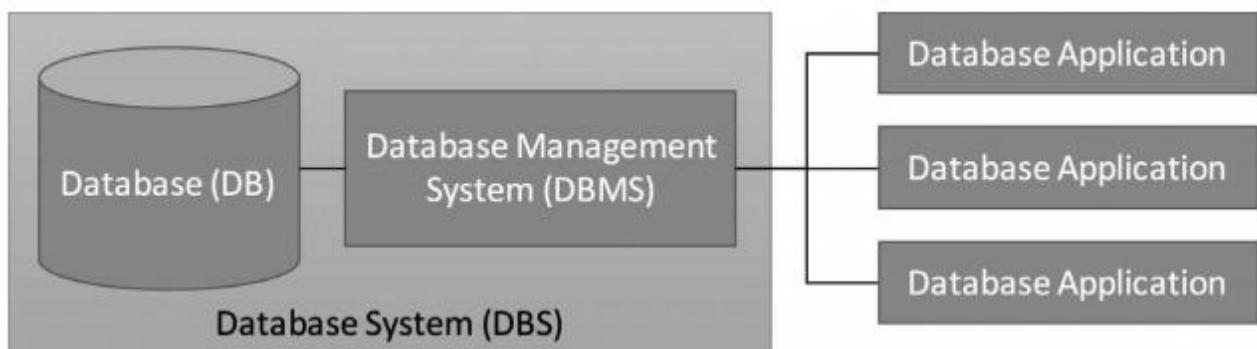


Figure 8: Database Management System

Knowing what a database is, it is important to know the **difference** between database and Database Management System [\[10\]](#) (from now on, **DBMS**). While a database is only the information itself, organized and collected in a specific way, a DBMS is a software that is used to manage that information, so that it is possible to capture the data, analyze it, use it on other applications, etc. This distinction is important, since nowadays those concepts are so closely related that whenever we speak of a database, we now may refer to data, to the traditional concept of database, or what is most common of all, a DBMS.

3.1. Taxonomy of DBMS

There are many different ways to establish categories for DBMS. More often than not, we see how they distinguish between SQL and NoSQL. However, this is just a lousy way to distinguish them, since it only refers to the fact that they may support the use of a **query language** called SQL or not. But there are many NoSQL systems that support SQL-like queries, so that classification will not be used in this work.

Relational

Non-Relational

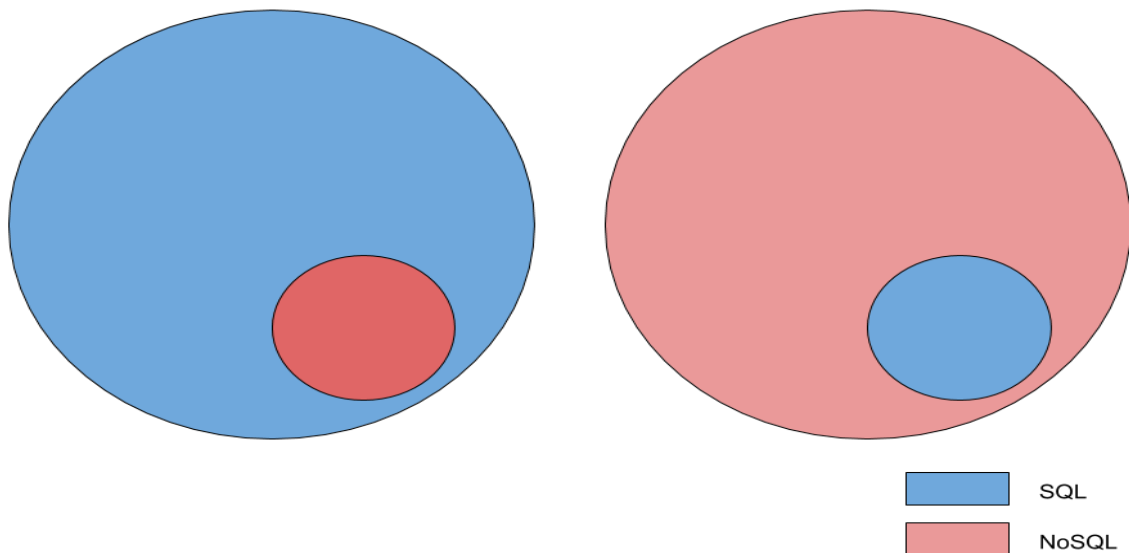


Figure 9: Diagram on Relational Vs Non-Relational and SQL vs NoSql

This SQL versus NoSQL is often mistaken with relational versus non-relational DBMS. Actually, the fact is that almost all relational databases support SQL, and almost all non-relational do not.

This mistake is in part because, when non-relational databases made their appearance, relational databases, which were pretty much the only ones in use, they all used SQL, so this generalization happened.

Therefore, in this work, we will not use such distinction, but we will address every kind of database according to its **database model**. However, according to a survey on non-relational databases, by Ali Davoudian and Liu Chen[\[3\]](#), these databases have a number of design characteristics in common:

- Otherwise than RDBMS, that support rigid schema-full [\[8\]](#) data models, non-relational are more flexible and often are schema-less, which allows data to have arbitrary structures instead being explicitly defined.
- They have relaxed the ACID properties that are part of a RDBMS, which allows them to scale out, achieving high availability and low latency, which are required by nowadays we applications, like *ShareCare*.
- They tend to place relevant data together in the same storage node and use data duplication, that way there is no need to be moving related data over the network all the time, achieving a better query performance. It means exchanging data duplication for speed.

- Most of them are designed to work in clusters, so data is easily replicated and horizontally partitioned.
- They count with simple client interfaces.

More and more, these non-relational databases are gaining in popularity¹.



Figure 10: DBMSs popularity chart

3.1.1. General-purpose DBMS vs Special-purpose DBMS

In an attempt to categorize databases, we could say there are **general-purpose DBMS** [7], which try to meet the needs of as many applications as possible, and **special-purpose DBMS**, that are used when we know the kind of data we have and we want to work with it in specific ways. Most of the general-purpose are RDBMS, and the special-purpose used to be non-relational, but there are cases like MongoDB, which tries to be general-purpose and is non-relational.

While general-purpose DBMS are often the most cost-effective approach, special-purpose DBMS are optimized to work in a certain way better than others: that is, they use their resources to make some operations more efficient than others. What they do, how and why is what we will study along this work, by means of studying different case of use of the web application *ShareCare*.

Examples of general-purpose DBMS are MariaDB or PostgreSQL, and examples of special-purpose DBMS would be Redis or ElasticSearch.

¹https://db-engines.com/en/ranking_categories

3.1.2. A better way to classify DBMS – Data model

Although the previous section already established a classification for DBMS, this way to differentiate them is still too broad. Next, there is a list of different families of databases, according to their data model:

- Relational DBMS (RDBMS)
- KV Store
- Document Store
- Graph
- Columnar
- DFS (not actually a DBMS, but still interesting to consider it within this study concerning data in web applications)

Traditionally, general-purpose DBMS are RDBMS, which still in the present day are the most widely used, and the others would be the special-purpose DBMS.

3.1.2.1. RDBMS

A **Relational Data Base Management System** is a database management system that works according to the relational model. Nowadays, most of the databases are based on this model.

As its name indicates, these databases are based on the so-called “relations” or “tables”. A table is a two-dimensional structure consisting of rows, also called tuples, and columns, also known as fields or attributes.

According to [\[1\]](#), chapter 4, in any kind of RDBMS, tables have the same characteristics:

- Each row is unique, so that information in one table cannot be duplicated. This helps avoid data redundancy.
- Each row in a table represents a single instance of an entity.
- The order each row appear in a table is unimportant, so rows are never identified by their position.
- Each column represents a unique attribute of the entity, and as it if for rows, their order is not important.
- All entries within a column have the same data type. That is, strings, integer, etc.
- Each cell contains a single value, that is, it may not contain a list of integers, or strings. It holds simple pieces of information.

- Each table should have only one primary key, whether this is consisting of one or more attributes of the table.

That primary key we mentioned above is what makes every different row unequivocally identifiable within the table. Although a primary key may be formed by more than one attribute, the **primary key should be minimal**, that is, it should consist of the minimum amount of attributes as possible.

There are also foreign keys, which are similar to primary keys, but they refer to the primary key of another table. As one of the rules establish, one cell may only contain a single value. Thus, using foreign keys is the way to make relations among tables, as if we were adding a whole instance of another entity as an attribute of one table.

Almost all RDBMS use a common language to make their queries and keeping the database contents up to date, which is SQL (Structured Query Language). This language is based on the standard ANSI/SQL, which defines its rules. However, depending on the specific RDBMS, they may use a dialect of this standard that implements what is established, other just implement a more relaxed version of it, such as MySQL, and others may implement more functionalities not mentioned in the standard.

The most widely known functions of this kind of databases are related to data creating, reading, update and deleting, or as they are commonly known, **CRUD** [\[14\]](#) operations. In order to carry out these types of operations, users usually use SQL, which allows to retrieve, store, modify, delete, insert and update data (using Data Manipulation Language) and create and modify the structure of database objects in database (using Data Definition Language) statements.

DML	
Function	Description
SELECT	Retrieves data from a table
INSERT	Inserts data into a table
UPDATE	Updates existing data into a table
DELETE	Deletes all records from a table

Table 10. Examples of DML

DDL	
Function	Description
CREATE	Creates objects in the database
ALTER	Alters objects of the database
DROP	Deletes objects of the database

Table 11. Examples of DDL

Relational databases follow a set of rules in order to operate efficiently and accurately. These rules are Atomicity, Consistency, Isolation and Durability, and they are known as **ACID**. And a series of operations that follow these rules is what is considered as a transaction.

Another key concept for RDBMS is **normalization** [\[15\]](#). This process uses the normal forms to restructure the database to reduce data redundancy and improve data integrity.

RDBMS can also support concurrency by means of using complex algorithms, which allows to maintain data integrity in a cluster.

Even though there are now plenty of alternatives to RDBMS when dealing with different ways to use the data, they are still the most prominent kind of database in the market. That means that there has been a great investment in their development ever since they were created, therefore, most of nowadays databases usages are still being carried out by them.

Examples of use cases

Apart from being mostly used as general-purpose databases, RDBMS are still among the best choices in the following scenarios:

- When the duplicated data would be huge if normalization does not happen.
- When dealing with low volumes of data and the need to carry out medium complexity operations (e.g., using a couple of joins of RDBMS instead of moving onto a graph database).
- When it is going to be more expensive in terms of money, time or complexity to migrate to other kind of system than it is to implement what you need in your current system.

Supermarkets or banks are still a good example of businesses who still use RDBMS.

3.1.2.2. KV Store

KV stores are used to **store an arbitrary value associated to a key**, which are stored in efficient and highly scalable key based structures, such as data hash tables. This way, as it happens with Python dictionaries, given a key, we can retrieve the value. That is why these are the simplest and most popular of all non-relational databases.

Since they are just key-value pairs, KV stores delegate all the responsibility of dealing with the value to the code itself, as it can be observed in figure 11.

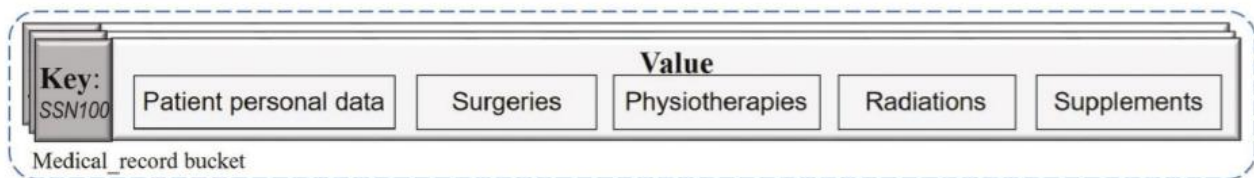


Figure 11: Representation of data in a KV store

Due to their extremely simple structure, they are really easy to partition, so that they work very well in clustered systems, and that is also why they are efficient to query data, which means they are **highly scalable**.

Opposite to how it happened in the previously mentioned RDBMS, where they had a specific schema that would define the kind of data that every table would contain, these KV databases treat every entry as a single item, regardless of what is inside. That is, for a given key, this could contain an integer, a string, etc., or, in cases such as Redis, lists or hashmaps.

In these databases, space is not wasted in fields that may not have any value, or may be optional, as it could happen with RDBMS, using way less memory, but at the same time that reduces organization, since the different entities that would belong to a same table in RDBMS do not have here that field standardization.

Regarding how these systems keep their information, some hold the data mainly in-memory, and whenever does not fit, they use disk, which is the case of Redis, and other send it straight to disk. As it happens with any other system to keep information in, accesses to memory are way faster than accesses to disk. So, in terms of data **persistence**, KV stores can be classified as:

- In-memory, such as Memcached, that are extremely fast by keeping the information in memory, suitable for application that deal with transient information that is going to last a limited amount of time. A good case of use for this would be storing the information of a user session in a website. This will be explained in the section [Examples of use cases](#).

- Persistent, storing the information in disk, what provides with high available access to the information, but they are not that fast. Riak KV is an example of this type.
- Hybrid, like Redis, as it was previously mentioned.

KV databases may provide some different levels of consistency. In some cases, they provide with the BASE semantics, meaning Basically Available, Soft estate, Eventual consistency. This means that, even if eventually the data may be consistent, requests may not get the updated value of a key, but they will always get a fast response.

Eventually consistent KV stores such as Voldemort or Riak use a method called *read repair*², which works as follows:

- When retrieving the value associated with a key, they decide which one out of several available values is the latest.
- If the most suitable value cannot be decided by the database, then a list with the possible values is provided to the client, who will pick one.

Examples of use cases

Using a KV store usually means that you have a priority for high-speed retrievals. In this cases, you can always afford take the risk of losing some data as a tradeoff for speed. Here there are a few examples of use cases for KV stores:

- Cache.
- Delivering web advertisements. Although these are essential for the companies advertising, that should not affect the users' experience when using that website. So, if advertisements make the rest of the website load more slowly, the users will find a faster alternative.
- Keeping track of an online shopping cart.
- Handling user sessions. KV stores are ideal for this purpose, since keeping a user session alive consists of having some kind of identifier that recognizes the user is still there. So two good points to use them for this purpose are that they will quickly retrieve the user information and that, as keys can be configured to expire in some time, whenever the key is not in the database anymore, the session is no longer valid, and the use will need to log in again.

²[\[6\]](#) page 98.

3.1.2.3. Document store

Whenever we need a flexible³ database, but we are going to be managing more complex data than those supported for KV stores (as we saw previously), we can turn to document databases.

A **document oriented** database is a kind of non-relational database in which information is stored in the form of a document. These documents refer to a hierarchical structure that may contain other substructures. As they are described by [6]:

“Documents can consist of only binary data or plain text. They can be semi-structured when self-describing data formats like JavaScript Object Notation (JSON) or Extensible Markup Language (XML) are used. They can even be well-structured documents and always conform to a particular data model, such as an XML Schema Definition (XSD)”

The same way that XSD is a group of specifications that specify how to describe elements in a XML, it is worth mentioning that JSON count with a similar specification document, the JSON schema⁴, which allows to annotate and validate JSON documents. They would be equivalent to the DDL of relational databases when, for instance, they create their tables.

Next, we can see examples for the same entity in XML and JSON documents.

Listing 1. Order XML Document

```
!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary><title>example glossary</title>
  <GlossDiv><title>S</title>
    <GlossList>
      <GlossEntry ID="SGML" SortAs="SGML">
        <GlossTerm>Standard Generalized Markup Language</GlossTerm>
        <Acronym>SGML</Acronym>
        <Abbrev>ISO 8879:1986</Abbrev>
        <GlossDef>
          <para>A meta-markup language, used to create markup
languages such as DocBook.</para>
          <GlossSeeAlso OtherTerm="GML">
            <GlossSeeAlso OtherTerm="XML">
          </GlossDef>
          <GlossSee OtherTerm="markup">
        </GlossEntry>
      </GlossList>
```

³This term may result controversial, as it will be explained later.

⁴<http://json-schema.org>

```

</GlossDiv>
</glossary>

```

Listing 2. JSON tree structure

```

{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such
as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}

```

Although it is usually said that these databases are schema-less, it is more appropriate to say that their schema is more flexible than others, since the data can be modeled more easily than it is, for instance, in relational databases, where DDL must be used. In Fowler's words, this flexibility "allows to load any kind of data without the database having the prior knowledge of the data's structure or what the values mean".

However, flexibility is a tricky term, since all kinds of databases may offer it to a certain extent. Document databases allow to leave DDL undefined, so they will infer the structure of the data themselves. For instance, looking at the example above, we could have defined the data structure as it follows:

Listing 3. Defining data structure of a document

```

{
  "mappings": {
    "glossary": {
      "properties": {
        "title": { "type": "text" },
      }
    }
  }
}

```

```
}  
  }  
}
```

In doing so, the field “title” is set to be of type text, and that could never be changed. As for the rest of the properties, they were not defined, so document databases create the structure itself, and once created, it will not be able to change either.

A document is the minimum unit of information, equivalent to what a row would be in a relational database. All those documents are grouped into **collections**, that way creating a hierarchy to access the data: collections consist of documents, which hold the information about “entities”, with all their different fields, whose types can be custom, and not necessarily fixed (i.e., integer, string, etc.).

Documents can also store other documents, which is known as **embedded** or nested document. This structure allows users to store related data into the same document, saving them the problem that in relational databases is solved doing joins, which they need to do to look up information in different tables. That operation can be costly on big relational tables.

Collections are used to operate on groups of documents that are related, so, for instance, it is possible to iterate through all the documents inside a collection. They also support additional data structures to make operations more easily, such as indexes, which map from an attribute to another related piece of information. It is easier to look for the key terms indexed instead of going through the whole collection.

As their schema allows to add new properties to documents, some of these kind may prioritize write availability over data consistency.

Apart from this, many document database use search engines, so they are widely used when fast query performance is of the essence. Also, using this search engines, they may implement functionalities such as text search and natural language analyzers.

Examples of use cases

These databases are really good to manage changing data structures. It could be used to store tweets, the short messages from Twitter. They are not only 140-character messages, but they have many fields behind that vary depending on the application that created the message. Other uses involve using a search engine to carry out a full text search within an application, as it happens in *ShareCare*, and will be later explained.

*Soysuper*⁵ would be a specific example. This is a website where you can do your groceries online, from a series of different supermarkets. They obtain the information of the products of every supermarket, and they store it with all their different fields. Here, every product will

⁵<https://soysuper.com/>

have their own fields, some may be common, and some others not. Also, there will be just so many fields to consider. On top of that, different supermarkets may have defined one same product in a different way. So, when *Soysuper* add these items, it does not know about all those fields beforehand, so a document database is a great asset for this purpose too.

3.1.2.4. Graph Databases

They are similar to relational DBMS, but unlike to most of the other non-relational databases, this kind of DBMS is based on a strong theoretical foundation.

Graph databases are a specialized kind of databases based on a branch of mathematics known as graph theory, which is really useful to analyze connections and links between entities.

A graph is a mathematical construct that consists of the following major components:

- **Vertices** or **nodes**.
- **Edges**, connecting vertices.
- Both elements can have **properties**.

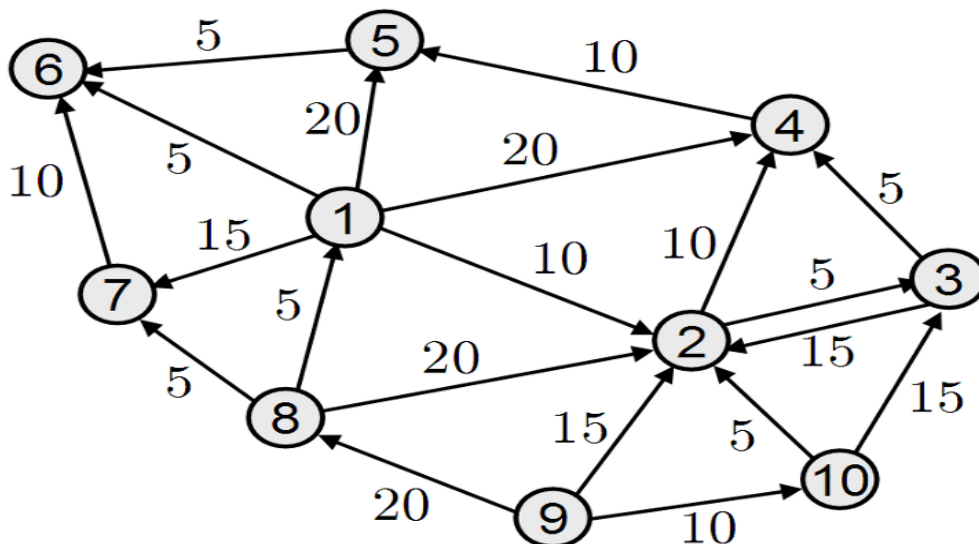


Figure 12: Example of a nodes and vertices in a graph

In graph databases, **vertices are a representation of any entity that has a relation with another entity**, and they are marked with a unique identifier. As the rows in a RDBMS, a vertex may have properties.

Edges represent the relation between vertices, and they may have properties too, such as the type of relation itself. A common property they have is **weight**, which may

represent some value about the relationship, such as cost, distance, affinity, etc., (as we have studied in the subject Data Structures). Also, edges may be **directed or undirected**, indicating that the relation applies, for instance, from one entity to another, but not vice-versa.

Also, there are two other concepts that are interesting to know regarding the **set of special operations** that graph databases are prepared for: paths and loops. Regarding those, there are three of these operations that are unique to graphs:

- **Union of graphs.** A union of two graphs is the set consisting of all vertices and edges of both graphs, the common and uncommon ones. If they have something in common, a single graph is produced, if not, the two graphs remain unconnected. This is useful to find, for instance, if a person is related to his friends' friends in a social network.
- **Intersection of graphs.** An intersection of two graphs gives as a result the group of vertices and edges that they both have in common. Following the previous example, this is useful to find what friends two people have in common.
- **Graph traversal.** Consists on visiting all vertices of a graph in a specific way. This is usually done to set or obtain a value in a graph.

According to [\[6\]](#), there are a set of operations that make graphs interesting as a database structure:

- **Shortest path:** finds the minimum number of hops between two vertices and the route taken.
- **All paths:** finds all routes between two vertices.
- **Betweenness:** given a set of vertices, returns how far apart they are within a graph.
- **Subgraph:** either finds a part of the graph that satisfies the given constraints or returns whether a named graph contains a specified partial graph.

When finding connections in a relational database, joins are required. As we have seen before, we use the foreign key in a table to look up a value from another table, which when dealing with large tables is very time consuming.

In words of Guy Harrison [\[5\]](#),

“Databases are about storing information about “things”, be those things represented by JSON, tables, or binary values. But sometimes, it’s the relationship between things, rather than the

things themselves, that are of primary interest. This is where graph databases shine”.

Therefore, graph databases are meant to be used when our main purpose is to **study the relations between entities**, and not the entities themselves. When working this way, the length of the queries could be unpredictable, because we would not know how deep the graph has to be traversed, and RDBMS lacks the syntax to easily perform such task. If we had to rely on a relational database to respond these kind of queries, it could take a really long time.

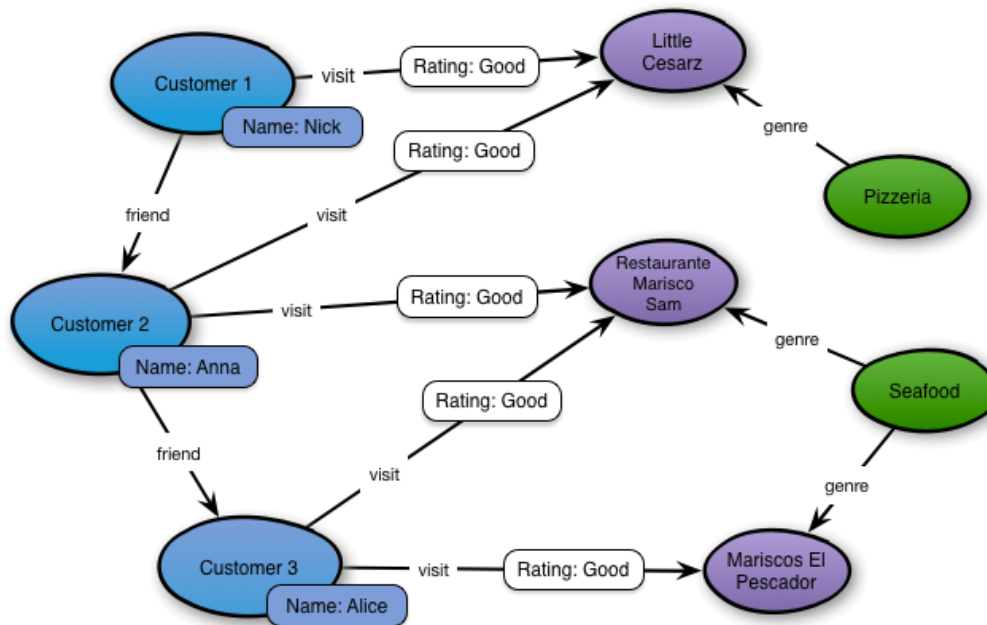


Figure 13: Example of relations in a graph

Customers		Rating			Restaurant	
id	name	id_customer	id_restaurant	rating	id	name
1	Nick	1	4	Good	4	Little Cesarz
2	Anna	2	4	Good	5	Restaurante Marisco Sam
3	Alice	2	5	Good	6	Mariscos El Pescador
		3	5	Good		
		3	6	Good		

Table 12. Representing a customer-restaurant relation in a relational database

As we can see, what in relational databases needs to be expressed as a many-to-many relation, which need the help of a third table (Rating), in a graph database is not

necessary, it is just enough to add a new edge. That is, **relations are explicitly modeled by the edges**.

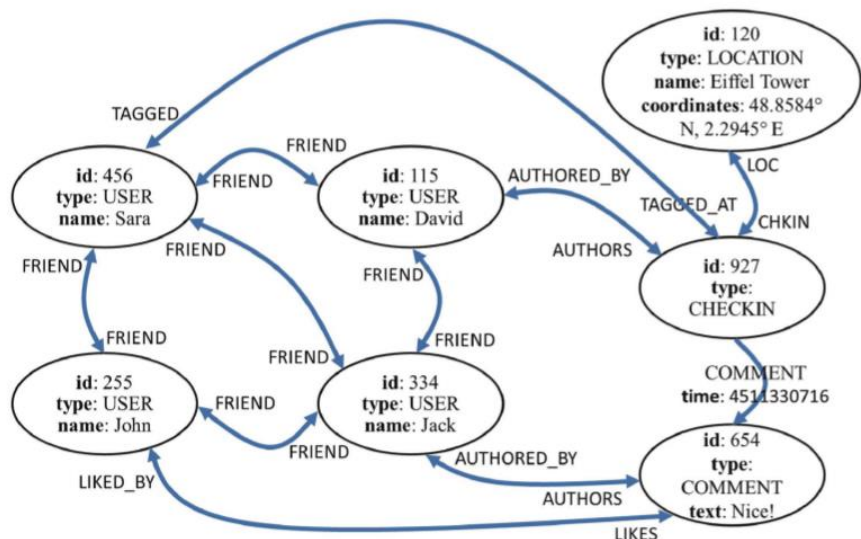
Examples of use cases

The most obvious use case for these kind of databases is the design of Social Networks. They are used the same way they are designed: to keep relations. But apart from using them to know how entities in a social network are related to each other (e.g. finding common interests of two people), they can also be used for [\[11\]](#):

- Recommendation engines. Retailers use them to spot patterns and make smarter recommendations.
- Fraud detection. Also, using pattern detection.
- Machine learning.

The following example, extracted from a work from Ali Davoudian, Liu Chen, and Mengchi Liu [\[3\]](#), illustrates very well how graph databases deal with relations:

Facebook's social network uses the graph data model. As an illustration, suppose David along with his friend Sara visit the Eiffel Tower. David uses his cellphone to record this visit by 'checking in' to the Eiffel Tower and tagging Sara to let other friends know that she is also there. Jack writes a comment on this and John likes it.



3.1.2.5. Column-wide Databases

As Guy Harrison notes, the conventional way to record our data was based on the Western culture, presenting rows that are written from left to right, up to bottom. That way it is understandable that the first data structures for databases were row-like, and they were stored as rows.

However, with the increasing needs for data warehousing and analytics, analyzing contents of rows (we already know that a row would contain an instance of an entity, with all its different fields, and different kinds of data) is not as interesting as processing the values

of a single column. That is why **column-oriented databases store columns physically together in disk**, to make these operations way faster.

The basic components of a column-wise database are keyspaces, row keys, columns and column-families.

- **Keyspace**: this concept is analogous to the relational concept of schema. All the rest of data structures created will be within a kespace.
- **Row key**: as a primary key does in RDBMS, it identifies a row in a column family, and also helps partitioning and ordering data. In column databases, rows are highly structured data items.
- **Column**: these are the equivalent to cells. They are where the values are stores. They consist of three parts:
 - Name. As a key in a KV store, it refers to the value.
 - Timestamp.
 - Value.
- **Column-family**: collection of columns that are related. It would be the counterpart of a traditional RDBMS table.

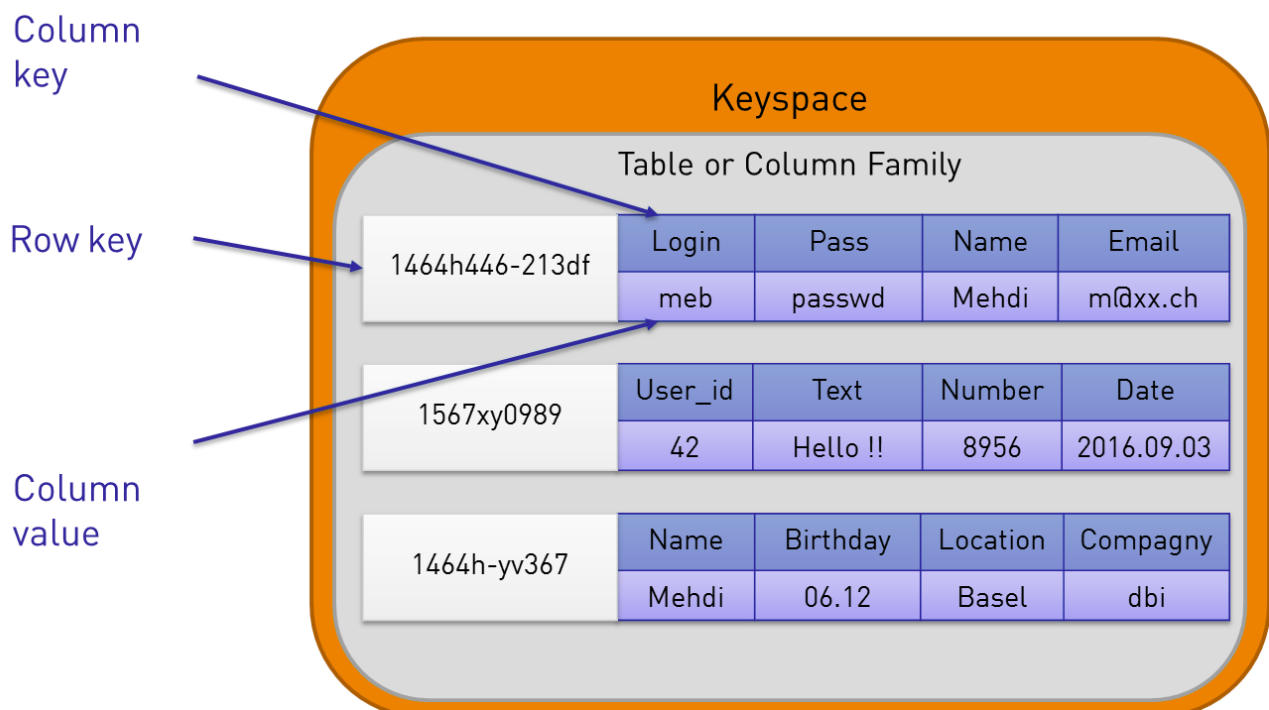


Figure 15: Example showing the hierarchy of the structure of a column database

While in RDBMS, as row oriented model, all columns for each row would be located in different disk blocks, in column-oriented databases, all the values of the same column

would be in the same block. This can be observed in the following example⁶, comparing the same database in row and columnar-oriented forms:

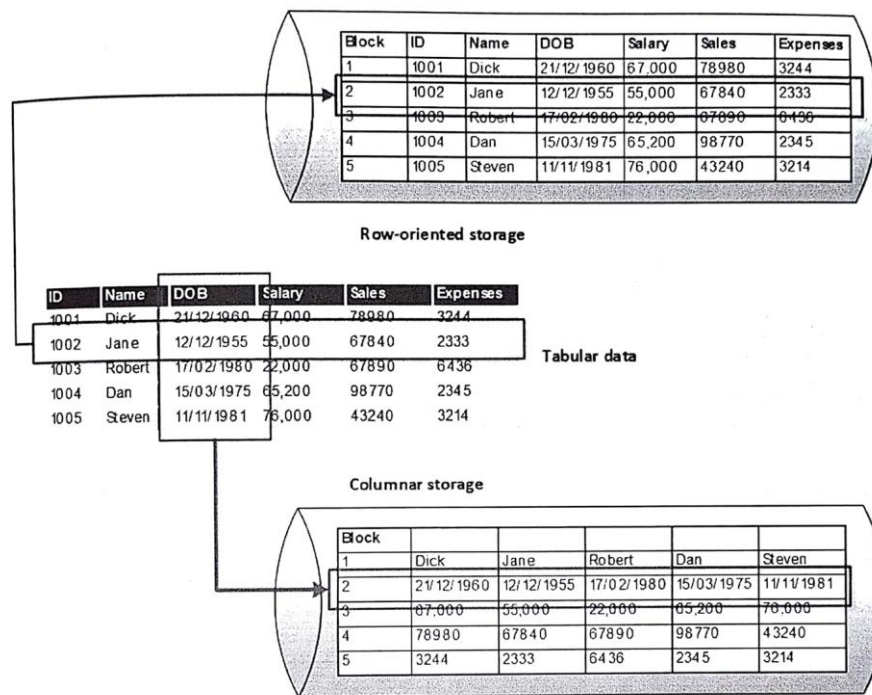


Figure 16: Comparison between row and columnar-oriented databases

Once we know the way the data are stored here, it is easy to realize how this comes in really handy in terms of space: **Compression**. As studied in the subject of *Multimedia Technologies*, compression is primarily based on data redundancy. Although the data inside the table will be the same, may it be in row or column form, when order in form of columns, the data that is more likely to be repeated is stored together, hence, compression algorithms will work way better. That is, **data will be more efficiently stored** in column-oriented databases.

As mentioned by in [5], the downside comes when with DML. For instance, when a traditional row-oriented database only need a single operation for inserting a row, column-oriented would need several accesses. That is, they do not perform well in terms of single data modification, that is, modifying all fields of an entity.

Examples of use cases

These databases are really good for data analytics. That is, whenever we need to analyze data of the same kind very quickly, like calculating the average temperature measured by some sensors in real time.

⁶Guy Harrison, Next Generation Databases, page 77

3.2. Databases information tables

This section is meant to show a couple of tables containing schematic general features about the different families of databases studied, as well as some performance information about the specific DBMS used in *ShareCare*.

Name	Primary model	Secondary models	Schema	Data type	Open Source	Triggers	Foreign keys	Durability	In-memory capabilities
Elasticsearch	Search engine	Document	No (semistructured)	Yes	Yes	Yes(Using percolation ⁷)	No	Yes	Cache
Redis	KV store	Document, graph, search engine	No	Partial	Yes	No	No	Yes	Works in memory
MariaDB	RDBMS	Document, KV store	Yes	Yes	Yes	Yes	Yes	Yes	Cache
AWS S3	DFS	No	No	No	No	Yes(AWS and Lambda)	No	Yes	Cache
Solr	Search engine	No	Yes	Yes	Yes	Yes	No	Yes	Cache
DynamoDb	KV store, Document	No	No	Yes	No	Yes(AWS and Lambda)	No	Yes	Cache
Couchbase	Document	No	No	Yes	Yes	Yes* (with TAP protocols)	No	Yes	Works in memory
PostgreSQL	RDBMS	Document, KV store	Yes	Yes	Yes	Yes	Yes	Yes	Cache
Mnesia	RDBMS	No	Yes (Dynamic fields)	Yes	Yes	No*(yes with pluggin)	Yes	Yes	Cache
Splunk	Search engine	No	Yes	Yes	No	Yes	No	Yes	Cache
FireBird	RDBMS	KV store	Yes	Yes	Yes	Yes	Yes	Yes	Cache
Memcached	KV store	No	No	No	Yes	No	No	No	Works in memory
Amazon Aurora	RDBMS	Document, KV store	Yes	Yes	No	Yes	Yes	Yes	Cache
GoogleCloud DataStore	Document	No	No	Yes	No	Yes*(Google App Engine callbacks)	Yes	Yes	Cache

Table 13. Features of different databases

⁷<https://www.elastic.co/blog/percolator>

Type of DBMS	Example	Writing frequency	Reading frequency	Max Sizes*	Data variety	Performance	Scalability	Flexibility	Complexity	Functionality	Structure
RDBMS	MySQL	Medium	Medium	64KB/row, 4092 rows per table	Types SQL and JSON	Moderate	Low	Moderate	Moderate	Relational algebra	Sequential rows on disk and hash or tree indexes
KV Store	Redis	High	High	512MB/String, collections of around 4 billion elements	Strings and other complex collections (Lists, Sets, Hashes...)	High	High	Low	Low	Variable (none)	Hash-tables, trees
Document	ElasticSearch	Medium	High	2GB/Document	Documents (JSON, XML, YAML, ...)	High	Variable (high)	High	Low	Variable (low)	Semi-structured
Graph	Neo4j	Medium	High	----	Numbers, Strings, Booleans, Spatial types, Temporal types	Moderate	Variable	High	High	Graphs traversing	Node and Edge table
Columnar	Cassandra	High	High	High	CQL types	High	High	Moderate	High	Minimal	Semi-structured
DFS	AWS S3	Low	Low	Unlimited	Multimedia Files	High	High	Low	Low	Minimal	Distributed Hash Table

Table 14. DBMS families overview

In these tables, we can see that many fields are related to the data, may it be its structure, its type or even the operations they may undergo. That gives us the notion of how knowing the data you want to work with is important in order to decide what kind of database you one should use. For instance, if our data is going to be accessed with a high frequency, and we do not need to make complex queries, we may want to pick some families, usually with a less fixed data schema, over others.

Even if we talk about data, that does not mean we are talking about some homogeneous entity. One same product may deal with a very wide variety of data, performing very different operations depending on the nature of the data. And as we are going to see in the next chapter, and as other well-known web application do, it is useful to use several different DBMS, because there is no single database system that excels at everything, as the tables show That is why it is necessary to study what is available in the market to make a good decision, and to reach a balance for what you need to do and how you and your selected systems are going to do it. And for that, it is also important to be up to date with this technology, to be able to realize if something you currently do can be done in a better, cheaper or less complex way.

Next, this information about database families will be applied to the context of *ShareCare*.

4. DBMS in *ShareCare*

Sharecare is a Q&A web application that works in similar way to StackOverflow.com. In this part, we are going to study four cases of use that have different preferences in terms of how the data are used, so different DBMS have been used.

The cases of use to exemplify the use of DBMS in *ShareCare* are the following:

- Storing user information
- Liking or disliking questions
- Uploading documents to complete their posts
- Making full text searches to find questions or answers that match what they are looking for

Each one of those features is going to use some data in a particular way. In this section, we are going to see what kind of DBMS is being used in every feature.

In order to know what kind of DBMS we need for a specific case of use, we ought to ask ourselves some questions, such as *“What kind of information am I storing?”*, *“What am I going to do with these data?”*, *“Do these data have a fixed structure?”*, *“Are these data going to be read or written with a high frequency?”*, etc.

For the purpose of this application and its cases of use, four different specific DBMS have been selected:

- MariaDB (RDBMS)
- Redis (KV Store)
- ElasticSearch (Document Store)
- Amazon Web Services Simple Storage Service (AWS S3. DFS)

Although the use of different kinds of special-purpose DBMS may not be necessary if the data on an application has special of some kind (millions of users accessing at the same time to some information, storing high volumes of data, etc), and obviously *ShareCare* is not an application currently running in the market, we will think of it as if it were something as stackoverflow.com.

This way, we can understand that every DBMS chosen has been selected to accomplish the tasks that does best.

4.1. Storing user information

Sharecare allows users to sign up, introducing their personal information. Up to this point, a user will have personal data and will keep record of the questions and answers that has posted and the votes for other users' contributions.

4.1.1. Using MariaDB

The data to be stored for the users may have the traditional structure of tables and relations. User data will be **structured**, that is, it will have some fixed attributes for their personal information, as their names and email, but they will be related to other entities, such as questions and answers, and that structure (or **schema**) is not likely to change, so we do not need the flexibility that other non-relational databases offer. Each user will have a one-to-many relation to questions, and another one-to-many relation to answers. MariaDB uses **primary keys** to identify the different rows of a table, and **foreign keys** that make possible to easily make relations between different tables.

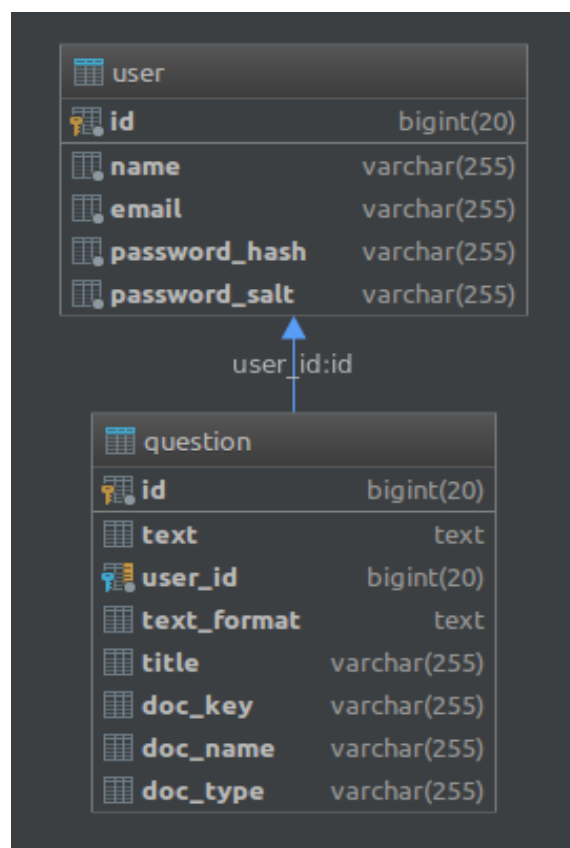


Figure 17: User - Question. One-to-Many relation

In the figure, we can see how questions and users are related. The entity Question has the attribute `user_id`, which is the foreign key that allows to make that connection, relating every question to one specific user.

This information is going to undergo CRUD operations, which refers to all the major functions that a RDBMS will do. Users will be created and deleted, and their information is going to be updated, but the data is not going to have a high frequency access or modifications. Users will not be changing their name, password or email with frequency. What is more susceptible to change within this context is the questions they ask and their answers, which represents **low volume** [13] of data in transactions and a **low frequency** of accesses.

Another good reason to use MariaDB is that, as a RDBMS, many big enterprises such as Google, Craigslist, Wikipedia, archlinux, RedHat keep using it nowadays, and that is reassuring. According to a post⁸ written by David Ramel [12], a survey commissioned by *Dell Software* in 2015 showed that “more than two-thirds of enterprises reported that structured data constitutes 75 percent of the data being managed”.

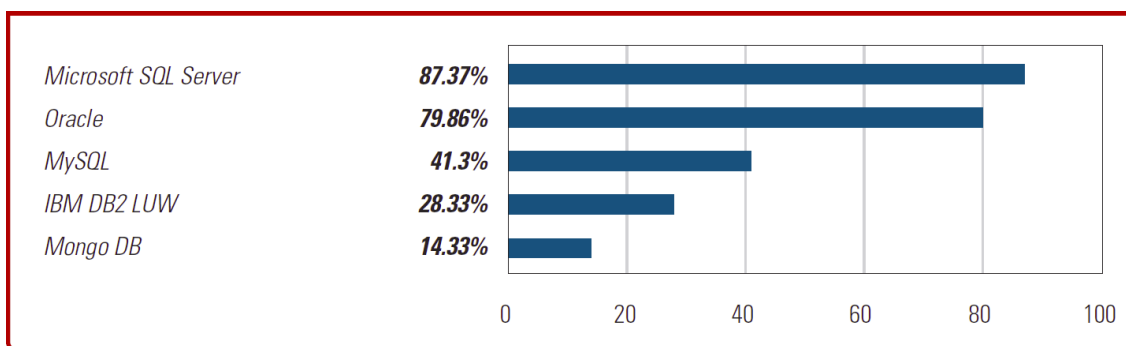


Figure 18: What Database Brands Do You Have Running in Your Organization? (source: Dell Software)

As a general-purpose DBMS, RDBMS are able to do many operations, although they may not be done the fastest way possible. However, speed is not the priority in this case, but the consistency and integrity of the data is prime, and that is one of the qualities that RDBMS offers, by putting the information in disk first. This is slower than using memory, but guarantees durability.

There are so many different RDBMS, so what are my reasons to choose MariaDB over another? To begin with, MariaDB is a fork of MySQL, with which I was familiar to more than any other. MariaDB is Open Source, and counts with a huge online community, there are plenty of sources of information, and as it has been running for long, there are just so many examples and kinds of applications that have worked with it before.

⁸<https://esj.com/articles/2015/04/23/database-survey.aspx>

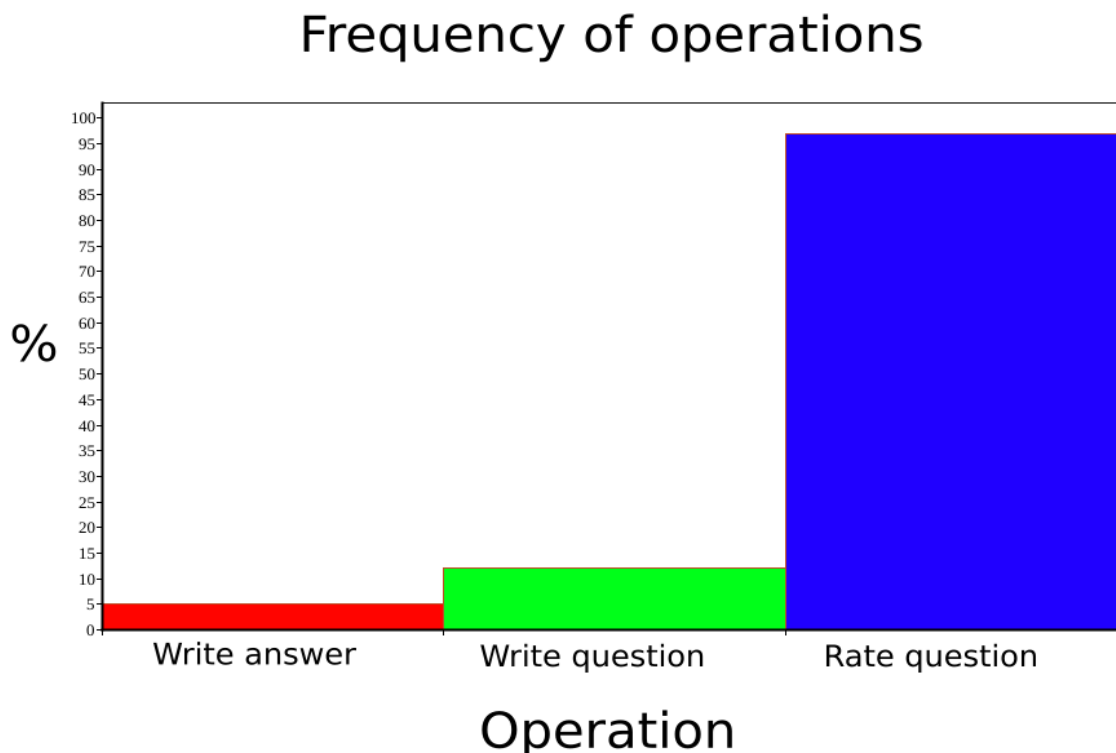
4.2. Liking or disliking questions

As it is commonly done in forums, users will vote for the questions and answers they find more useful or interesting. Although it is not implemented so far in *ShareCare*, this rating would also be used as an important factor when searching.

4.2.1. Using Redis

Votes for questions and answers are simple represented by a number. If a user likes a question, clicking the proper button will make this number increase. This is what users do the most, rate a question or an answer. As we can see in this kind of applications, we may find a question that counts with 20 or 30 answers perhaps, but the number of likes and dislikes may be of thousands. This means we need a DBMS that is suitable to work properly with a **high frequency** of readings and writings.

In these Q&A web applications, users write questions and answers, but what is most common of all is to rate questions, so there will be way more rating than anything else.



KV Store DBMS are really fast because usually they work in **memory** and their structure is really simple. They do not make complex operations, and for the purpose of the

case of use, the only thing we need to do is to increase and decrease a counter. However, these operations will be taking place very frequently.

As the data involved in this is just a number, we do not need to keep a rigid schema. Thanks to Redis' very simple data structure, and the fact that it works against memory first, this case of use is a perfect match for it.

Key	Value
likeQuestionCount:80	"1"
dislikeQuestionCount: 83	"0"
viewcount:83	"3"

Table 15: Example of contents in Redis

As we can see in the example, all we need to store is some numerical values to increase them. Although we would need a key and also the id number of the element, as there is no structure and we can build every item the way we want, we will use a name space constituted by that key name we give it together with the id number of the entity used in MariaDB for that item.

However, getting (reading) and setting (writing) values are still two different operations necessary to update any value. Now, with Redis **it is possible to do both in one only atomic operation**, using the command INCR (or DECR). That is, let's say two users click the button like of the same question at the same time. Usually, the problem we face is that user one gets the value and before updating it, user two gets the same value. Then, they both increment the value, and it turns out that the value increased in 1 instead of 2. That is what means that INCR is atomic: the read-increment-set operation will only be performed while none of the other users are executing that command at the same time.

Of course, the need for special-purpose DBMS such as Redis needs to be taken into consideration once looking into the future, when you deal with real world data and you really need to process millions of these requests per second.

4.3. Uploading documents to complete their posts

As a Q&A application devoted to help health professionals diagnose their patients, many times it may turn out useful to upload pictures or document to back up their questions or answers. This means that potentially every question will have some kind of document attached.

4.3.1. Using AWS S3

In this case, the choice was not exactly a DBMS, but a DFS (Distributed File System). Files are usually going to be a lot heavier than the rest of information we store in databases. When storing these files, we do not need any relation between them. Every file is totally independent from each other, the modification or deletion of one does not affect at all to the others. So, all we need for the system is to be **secure**, **durable**, and **highly scalable**. We need to be able to store huge amounts of information, since we will store multimedia files that may be really heavy, and not just simple data types.









<input type="checkbox"/>	Name 	Last modified 	Size 	Storage class 
<input type="checkbox"/>	 1804b853-5a1b-41ce-94a8-292cc3a6bda5	Jun 7, 2018 9:41:05 AM GMT+0200	77.4 KB	Standard
<input type="checkbox"/>	 1fbef31-2a03-490a-924d-49c254063d35	Jun 7, 2018 12:08:35 PM GMT+0200	77.4 KB	Standard
<input type="checkbox"/>	 32ee2775-e418-45cc-898e-6170acdd79ea	Jun 7, 2018 9:39:12 AM GMT+0200	1.9 MB	Standard
<input type="checkbox"/>	 4eaf050a-af46-4974-b87e-b6ddc5fc756f	Jun 7, 2018 9:28:53 AM GMT+0200	1.9 MB	Standard

Figure 20: Miscellaneous files stored in bucket ShareCare in S3

As we see in the figure, what is stored in the buckets of S3 are files with no relation. What is necessary to allow other users who read a question to see those uploaded files, is to configure the service online to allow any user to see the content, and also set the access as public when uploading the file using the java API that AWS provides with.

Although it is possible to store files in DBMS, it is not recommendable. Not only because of the size of the files, that could be huge, but also because of the range of sizes they may have. For instance, we could try to store files using MariaDB. And if we are uploading pictures that are a few KB heavy, that would not be a bad idea. MariaDB has a data type called “blob”, which stands for **Binary Large Object**, intended for this purpose among others. But the thing is that MariaDB uses a physical structure called page to store the information in disk, and the size of these pages is 64KB. That is, ideally a row in a table would be at most 64KB heavy.

4.4. Making full text searches

User visit this website to make consults about their doubts, but every person writes in a different way. When we use other tools such as web browsers like Google, we ask our question and we obtain a relation of results that match, even if not all the words we wrote appear, or not in the same order. Google check the content of all relevant text inside every website, applying its famous algorithm PageRank, and returns the results from best to worst.

4.4.1. Using Elasticsearch

We already had questions and answers stored in MariaDB, with their text bodies, titles, and the content with their html tags. So we do not need a storage to keep all that information, we just need all the text of every element put together. **A solid structure is not necessary to perform this task.** In this case, we do not want to make a distinction between the contents of questions or answers, all we want is their unique item ids and their content. That is, we are going to use a schema that allows a **faster way to operate with the data**, we do not need a rigid structure. Elasticsearch makes this task of searching through a pool of questions really fast and **easy to integrate by the developer**.

As studied before, in the section *Document store*, these documents can be, for instance, JSON or XML. Elasticsearch uses the first one for its documents, and according to [4], they are constructed as it follows:

- Data is organized in key-value pairs, similar to KV stores.
- Documents consist of name-value pairs separated by commas.
- Documents start with a "{" and end with a "}".
- Names are strings, such as ["all_content"] and ["className"].
- Values can be numbers, strings, Booleans (true or false), arrays, objects, or the NULL value.
- The values of arrays are listed within square brackets.
- The values of objects are listed as key-value pairs within curly brackets.

Next, we can see an example of a query made to Elasticsearch and its response, which is a JSON. We are requesting all the documents stored in the index "questions":

Listing 4. Elasticsearch search request matching all

```
curl -X GET "localhost:9200/questions/_search?pretty" -H 'Content-Type: application/json' -d '{
  "query": { "match_all": {} }
}'
```

Listing 5. Elasticsearch search response

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
```

```

    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "questions",
        "_type" : "question",
        "_id" : "4AL5ymMBHh4iaEEwncbG",
        "_score" : 1.0,
        "_source" : {
          "all_content" : "El paciente no tiene nada de nada Paciente con tos y sin
mocos",
          "className" : "Question",
          "entityId" : 81,
          "body" : "El paciente no tiene nada de nada",
          "title" : "Paciente con tos y sin mocos"
        }
      },
      {
        "_index" : "questions",
        "_type" : "question",
        "_id" : "3wL5ymMBHh4iaEEwRsau",
        "_score" : 1.0,
        "_source" : {
          "all_content" : "El paciente tiene resfriado y toas. Paciente con resfriado y
tos",
          "className" : "Question",
          "entityId" : 80,
          "body" : "El paciente tiene resfriado y tos.",
          "title" : "Paciente con resfriado y tos"
        }
      },
      {
        "_index" : "questions",
        "_type" : "question",
        "_id" : "4gL4z2MBHh4iaEEwssbe",
        "_score" : 1.0,
        "_source" : {
          "all_content" : "Paracetamol y reposo ",
          "className" : "Answer",
          "entityId" : 9,
          "body" : "Paracetamol y reposo",
          "title" : ""
        }
      }
    ]
  }
}

```

This example shows that there are 3 documents in the index “questions”. As there was no search, all the documents in the index were return as result.

Now, in order to make a full text search, Elasticsearch allows you to use a natural language analyzer and customize it the way you want. For this case, the analyzer was created to identify Spanish language, not paying attention to words that have no semantic meaning (those will be known as **stop words**, such as prepositions and conjunctions), and will also recognize the **stems** of the words, so it will not matter if the user is searching “*resfriado*”, “*resfriada*” or “*resfriados*”. Let’s see a example of that:

Listing 6. Elasticsearch search request matching specific content

```
curl -X GET "localhost:9200/questions/_search?pretty" -H 'Content-Type: application/json' -d'
{
  "query": { "match": { "all_content" : "resfriada" } }
},
'
```

Listing 7. Elasticsearch search response to specific matching

```
{
  "took" : 3,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.90232176,
    "hits" : [
      {
        "_index" : "questions",
        "_type" : "question",
        "_id" : "3wL5ymMBHh4iaEEwRsau",
        "_score" : 0.90232176,
        "_source" : {
          "all_content" : "El paciente tiene resfriado y tos. Paciente con resfriado y
tos",
          "className" : "Question",
          "entityId" : 80,
          "body" : "El paciente tiene resfriado y tos.",
          "title" : "Paciente con resfriado y tos"
        }
      }
    ]
  }
}
```

```
}  
}
```

We have now made a search, looking inside the field “all_content” of the documents inside the index “questions”. As we can observe, searching “*resfriada*” get the results containing the stem of that word, finding matches containing the word “*resfriado*”.

5. Deployment

In this section we are going to see what is necessary to have this project up and running for development. *ShareCare* has been developed on Ubuntu 16.04 LTS, and as mentioned before, it uses Java 8.

The first part of this section is going to deal with the installation of the environment. We will see what we need to install, including the databases involved. After the installation of the databases, we will see how to start running them.

In the second part we will see how to generate the JAR file, ready to upload to the server and start running it.

5.1. Preparing the environment

Installing Maven

To build the project, it is necessary to have Maven version 3.3.9 installed. To do so, we just need to open a terminal and use the following commands:

Listing 8. Install Maven 3.3.9

```
$ apt-get -y install maven
```

Installing Node

We will use Node.js to run Javascript on the server. We also install npm (node package manager). The installed version was the stable versions at the time, 8.10 and 5.6 respectively.

Listing 9. Install Node.js and npm

```
$ sudo apt-get install nodejs  
$ sudo apt-get install npm
```

Before that, we may need to update our local repositories

Listing 10. Updating local repositories

```
$ sudo apt-get update
```

Installing Gulp

Once Node is install, we use npm to install gulp

Listing 11. Install gulp

```
$ sudo npm install --global gulp
```

The dependency for gulp is already within the project, so we do not need to do anything else about it now. It was easily added thanks to using Maven.

Starting databases

After installing all this, our machine will be able to start running the software. However, we still need to start our DBMSs.

MariaDB

We need to install our MariaDB server and client, version 10.2.13.

Listing 12. Install MariaDB

```
$ sudo apt-get install mariadb-server mariadb-client
```

And then, to start running it

Listing 13. Start MariaDB

```
$ sudo systemctl start mysql
```

Now, to add the database structure with its tables, we do the following:

Listing 14. Opening MariaDB terminal

```
$ mysql -u [username] -p[password]
```

Once inside MariaDB, we need to create the database first.

Listing 15. Creating and using database

```
$ CREATE DATABASE sharecare;
```

Redis

Redis provides in its website the commands to download it and compile it really easily. We will use Redis 4.0.9.

Listing 16. Download and install Redis

```
$ wget http://download.redis.io/redis-stable.tar.gz
$ tar xvzf redis-stable.tar.gz
$ cd redis-stable
$ make
```

After this, we just need to navigate to the directory where we installed it (if you just followed the example, it will be in downloads) and start the server.

Listing 17. Starting Redis server

```
$ redis-server
```

We do not need to start a client, our application will be starting the client, using Jedis. However, we can start a client in the console so that we can verify whenever we want that the values are being written in Redis. To do so, we just need to run redis-cli in the terminal.

ElasticSearch

Similar to Redis, we need to download ElasticSearch and run the server in our machine. We just need to follow the instructions in the ElasticSearch website to get the version 6.2.4.

We first download it using curl

Listing 18. Download ElasticSearch

```
$ curl -L -O https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.2.4.tar.gz
```

Then we extract the tar:

Listing 19. Decompress file

```
tar -xvf elasticsearch-6.2.4.tar.gz
```

After this, we can start our ElasticSearch node executing the script within the directory “bin” in the extracted folder.

Listing 20. Start ElasticSearch

```
$ ./elasticsearch
```

Amazon Web Services S3

In order to use AWS S3, it is necessary to create an account in <https://aws.amazon.com/>. Once the account is created, we need to go to IAM (Identity and Access Management) to create a user. This user is what we will use to connect our application to AWS S3. Here, we need to create the user, select the option programmatic access to allow the connection through an access key and a secret key, and then we will give it the permission “AmazonS3FullAccess”.

After creating this user, we need to go to S3, under the tag Services, and create a bucket. A bucket is the specific storage where our information will be stored.

Now, we just need to use the Java API provided to access the storage, using both keys and the bucket name.

5.2. Starting *ShareCare* for the first time

5.2.1. Creating JAR

First, we need to download the project from the GitHub repository, either downloading the zip or using *git clone*.

Listing 21. Downloading zip

<https://github.com/yockiehub/sharecare/archive/master.zip>

Listing 22. Cloning project

```
$ git clone https://github.com/yockiehub/sharecare.git
```

After downloading and extracting the files, we need to build the project, generating the JAR file. To do that, we need to run the following command:

Listing 23. Building the project

```
$ mvn package
```

Maven projects have a folder called *target*, where the resources are generated, so we will find our JAR file there. But there are two JAR files in there. One is the original JAR, including just the main class. However, the project includes a lot of dependencies, so we would need to add them one by one when building the project. But instead of doing that, Maven generates another JAR, which already includes all the dependencies. They are easily distinguishable for two reasons: the first file has a suffix “.original”, and is not heavy, while the second does not have that suffix and is way heavier, since it contains the information of all those dependencies.

So the smart option is to go to the *target* folder and execute the following command in the terminal:

Listing 24. Executing JAR

```
$ java -jar target/sharecare-platform-webapp-0.0.1.jar
```

But if we do this, we will have a compilation error, since we are missing some parameters that are needed to connect to Amazon Web Services S3. So the command we need to actually run the application on the server is the following:

Listing 25. Execute JAR with AWS S3 keys

```
$ java -jar target/sharecare-platform-webapp-0.0.1.jar --aws.key=[ACCESS_KEY] -  
aws.secret=[SECRET_KEY]
```

Here, ACCESS_KEY and SECRET_KEY are the keys provided by AWS S3 (which are personal and will not be shared here). How to obtain these keys will be explained in next chapter, in the section 6.4.4. *Amazon Web Services – Simple Storage Service (AWS S3)*.

5.2.2. Creating data structure for MariaDB and ElasticSearch

In section [5.1. Preparing the environment](#), we had our MariaDB database create, but now we need to create the tables. As mentioned before, we will just need to copy the content of “migration_1.sql”, found in the root of the project, into the terminal.

Another way to do it is just dumping the content of the file into MariaDB using the pipe, as it follows:

Listing 26. Adding tables to database

```
$ mysql -u [username] -p[password] sharecare < migration_1.sql
```

As for ElasticSearch, once it is up and running we need to create the index and the text analyzer for the full text searches. For that purpose, there is a file called “ESmapping” within the project. We just need to copy and paste its content in a terminal. After that, everything is ready to start using *ShareCare*.

6. Technologies involved

There are many different kinds of technologies involved in the creation of a web application such *ShareCare*, may them be IDEs, frameworks, plug-ins, task managers and project builders, as well as, obviously, different kinds of databases.

The application back-end has been developed using Java 8, which was specifically selected in order to be able to use needed versions of other technologies that we will see later on this section, such as ElasticSearch.

With respect to the tools used to help build the project, Maven will be the chosen project builder.

As this is a web application, we will make use of HTML for the views, CSS to define the style and Javascript to deal with the client side of the application.

6.1. Integrated Development Environment – IntelliJ IDEA

For Java development, this IDE is one of the best options now out in the market. Along the whole degree, we have used several different IDEs to develop in Java, such as NetBeans, or Android Studio when dealing specifically with that platform. Actually, this last one was developed by the same labs that made IntelliJ IDEA⁹.

This IDE was a great choice since, in its premium version, gives you lots of help in order to integrate different frameworks, such as Spring Framework, that I will be using. Apart from this, it includes many tools to make navigation through the project really easy, auto-complete, etc.

6.2. Spring Framework

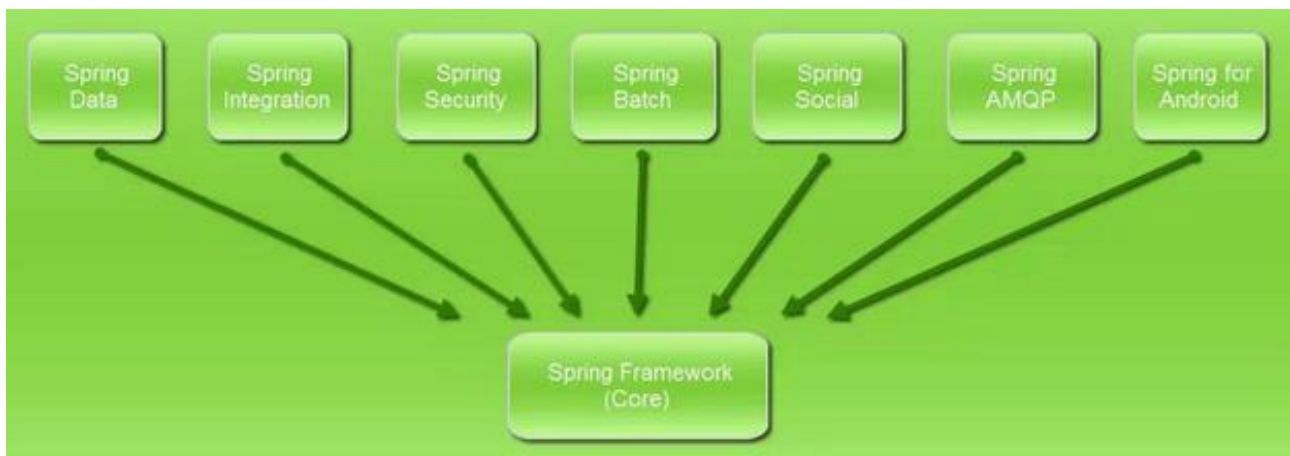
Spring Framework¹⁰ is a set of completely independent modules that can be used according to the needs of every project. It deals with all kinds of complicated configurations, while the user only needs to use some annotations and let it do all this obnoxious job. This is called inversion of control (IoC).

⁹<https://www.jetbrains.com/idea>

¹⁰<https://spring.io/>

When developing any kind of software, it is rare not to use external libraries, or frameworks that deal with repetitive tasks or inject some code that is annoying to be writing all the time. Usually, it is the developer who needs to configure all the different libraries and frameworks so that they can all work together within the application. So, using frameworks is not strictly necessary when starting to work on a project, although it is extremely useful and nowadays it is rare to find a project that does not count with the help of one. Without it, developers would need to look or implement every little thing, including some utilities that may have already been efficiently implemented by someone else, and find the libraries where they are in and add them to the project themselves.

However, Spring Framework is, what we could call a “framework of frameworks”. By means of using annotations, it will create all the different necessary objects that the user would need to do one by one, so that the developer can focus on the important aspects of the code.



Spring Framework consists then of a series of modules that provide with different services. Among those, the most relevant for this project are the following:

- Spring Core Container: this is the base module for Spring, provides with BeanFactory and ApplicationContext.
- Authentication and authorization: configures the security of the application, thanks to the module Spring Security. This will be used to create a secure way for users to sign up and register.
- Data access: tools to work with relational databases using JDBC (Java Database Connectivity), as well tools to work with non-relational databases.
- Inversion of Control Container: this configures all the application components and has control over the life cycle of Java objects, mainly through dependency injection. That is, how all these frameworks are integrated and how they work with each other is done by means of this Inversion of Control, which consists on taking the control of

the application from hand of the user, perform a series of tasks, injecting the necessary code (depending on the Spring annotations used), creating beans and managing the components, and then giving the control back to the user.

6.3. Maven

Maven¹¹ is a software project manager and dependency manager, it makes adding dependencies to a projects easier than just adding them one by one by yourself. Maven uses a configuration file called *pom.xml*, in which the reference to the dependencies are added, and then Maven will find them and import them into the project automatically.

Among the tasks that Maven can do to help the developer:

- Build the Java project.
- Execute testing tasks.
- Keep a version control.
- Deploy project in a server.
- Create documentation in HTML

6.4. Integrating databases

As it has already been said, the back-end of the application is written in Java. In this subsection we are going to have an overview of how these technologies were integrated into the project. All this needs to be done after successfully completing all the steps in section 5.1. *Preparing the environment*.

6.4.1. ElasticSearch

ElasticSearch¹² is the Document database and search engine that is going to be used to perform full text searches. There are two main steps to follow:

1. Include the dependency into Maven's parent *pom*.

Listing 27. ElasticSearch dependency for Maven

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>transport</artifactId>
  <version>6.2.4</version>
```

¹¹<https://maven.apache.org/>

¹²<https://www.elastic.co/products/elasticsearch>


```
</dependency>
```

2. Use that dependency to create a client for ElasticSearch. See **ElasticSearchConfig.java** and **SearchService.java** for more details about the implementation.

6.4.2. Redis

About Redis, we have a wide variety of Java APIs. Among those, the chosen one was Jedis¹³, and it is really easy to use. Steps followed for the integration of Jedis:

1. Include Jedis dependency in Maven

Listing 28. Jedis dependency in Maven

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```

2. Creation of a configuration file to create and start Redis client. See **RedisConfig.java** and **LikeService.java** for more details.

6.4.3. MariaDB

For MariaDB¹⁴, the configuration can be found within the parent configuration file **application.yaml**. There, it is necessary to specify the user and the password used to start MariaDB server as it was explained in the previous chapter. Also, as with the other databases, we need to add the dependency.

1. First, we add the dependency in Maven.

Listing 29. Adding MariaDB dependency

```
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
  <artifactId>mariadb-java-client</artifactId>
  <version>1.5.9</version>
</dependency>
```

2. Add settings to establish the connection in **application.yaml**.

¹³<https://github.com/xetorthio/jedis>

¹⁴<https://mariadb.org/>

Listing 30. Configuring access to Maria DB

```
jpa:
    hibernate:
        ddl-auto: none
datasource:
    url: jdbc:mysql://localhost:3306/sharecare
    username: [username]
    password: [password]
```

Where [username] and [password] are the user name and the password to log in our database, as we did in the previous chapter, when installing MariaDB.

3. Create repositories for the entities. Check all classes ending in “Repository” for more details.

6.4.4. Amazon Web Services – Simple Storage Service (AWS S3)

As for AWS¹⁵ S3, we need to use the Java API Amazon offers to establish a connection with the server using the data we obtained when we created the account, but we also need the appropriate dependency.

1. First we include the dependency.

Listing 31. Adding AWS S3 dependency

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk-bom</artifactId>
    <version>1.11.339</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

2. Creation of the configuration file **AWSConfig.java**, where we will create the client and connect to the server using the keys obtained after creating the account. Then we use the API methods provided by Amazon to upload and download content from the bucket that was previously created. Check **DocumentService.java** for more details.

¹⁵<https://aws.amazon.com/>

7. Conclusion

Talking about software means talking about information, data, and that needs to be stored somewhere, somehow. Although databases had originally nothing to do with computers, since they existed from way before this technology, now it is not possible to conceive any informatics system without them.

No matter what kind of application we are trying to develop, in one way or another we are going to need to use some DBMS. However, some of them are better than the rest on doing a specific task.

When we are developing some software, we need to know what kind of data we are going to use, and what kind of operations are going to undergo those data. **Knowing your data is one of the most important things** you need to know in order to choose what kind of DBMS you will be using. As seen along chapter 4, every database was chosen for some feature they have that make them work with the specific kind of data in a better way than others may do.

When the workload of the application is not really high, it usually does not matter whether you are using special-purpose DBMS or just a general-purpose database, since nowadays many DBMS can handle most of the operations a web application may deal with. However, the real problem comes when an application scales and it starts dealing with real data, like having millions of users, accessing data with high frequency, etc.

So now, the key to working with your data more efficiently is to **find a balance between the use of as many different DBMS that you would need and the cost** of using them, may it be monetary, time-consuming, or how troublesome it is. The tables at the end of chapter 3 may be helpful to find that balance.

Often, we may run into some enterprises that only use one kind of database for everything: even if it is been used for purposes it was not intended, it may be a good choice if your data treatment allows it. It will make the software easier to develop and it will not be necessary to learn new and more complex stuff to maintain the database.

As we have seen along this work, there are plenty of databases of many different types. And as needs grow, more and more DBMS appear offering your business new possibilities. Therefore, the **need to stay updated and aware of the different platforms** in the market is of utmost importance.

There is no exact method to choose the perfect database infrastructure now for any kind of software, for any kind of data or for any moment in time. Technologies evolve, data grows, and new uses and ways to work appear as society moves forward. However, the line

of this work may serve as a basis to create that methodology, to help the decision of what kind of DBMS use for diverse specifications we may think of.

In terms of the application *ShareCare*, there are plenty of possibilities. The web application could become a professional work network, where the centers where the doctors that use this application work can manage groups, create events to broadcast to all their doctors, for instance. Ideally, this would be applied world-wide, so that knowledge could be more easily spread to every country. For that, it would be necessary to deal with translations, and how to store the information on different languages in a database efficiently.

8. References

- [1] Eckstein, Jonathan, and Bonnie R. Schultz. *Introductory Relational Database Design for Business, with Microsoft Access*, John Wiley & Sons, Incorporated, 2017. ProQuest Ebook Central, <http://ebookcentral.proquest.com/lib/htwg-konstanz/detail.action?docID=5131970>. Created from htwg-konstanz on 2018-06-14 05:45:16.
- [2] Harrington, Jan L.. *Relational Database Design and Implementation: Clearly Explained*, Elsevier Science & Technology, 2016. ProQuest Ebook Central, <http://ebookcentral.proquest.com/lib/htwg-konstanz/detail.action?docID=4509772>. Created from htwg-konstanz on 2018-06-14 06:04:50.
- [3] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. *A Survey on NoSQL Stores*. ACM Comput. Surv. 51, 2, Article. 40 (April 2018), 43 pages. <https://doi.org/10.1145/3158661>
- [4] Sullivan, Dan. *NoSQL for Mere Mortals*. Addison Wesley. Michigan, USA, April 2015. *For Mere Mortals Series*. ISBN-10: 0-13-402321-8.
- [5] Harrison, Guy. *Next Generation Databases*. Apress. New York, USA, 2015. ISBN-13: 978-1-4842-1330-8.
- [6] Fowler, Adam. *NoSQL for DUMMIES*. John Wiley & Sons, Inc. New Jersey, USA, 2015. ISBN: 978-1-118-90574-6.
- [7] Wikipedia, <https://en.wikipedia.org>, *Database*. <https://en.wikipedia.org/wiki/Database>. Date of retrieval: March 23, 2018.
- [8] TutorialsPoint, <https://www.tutorialspoint.com>, *DBMS – Data Schemas*. https://www.tutorialspoint.com/dbms/dbms_data_schemas.htm. Date of retrieval: April 27, 2018.
- [9] <http://www.infosysblogs.com>, Infosys, *Key - Value store vs. Relational database in Cloud context*, Akansha Jain, May 31, 2010. http://www.infosysblogs.com/cloud/2010/05/k-v_store_vs_relational_database_in_cloud_context.html. Date of retrieval: May 3, 2018.
- [10] Maestros del Web, <http://www.maestrosdelweb.com>, *¿Qué son las bases de datos?*, Damián Pérez Valdés, October 26, 2007. Date of retrieval: April 25, 2018.
- [11] Computer World Uk, <https://www.computerworlduk.com>, *Seven enterprises using graph databases: Popular graph database use cases, from recommendation engines to fraud detection and search*, Scott Carey, May 11, 2017. <https://www.computerworlduk.com/galleries/data/7-most-popular-graph-database-use-cases-3658900/>. Date of retrieval: May 5, 2018.

- [12] Enterprise Systems Journal, <https://esj.com>, *Relational Databases Still Reign in Enterprises, Survey Says*, David Ramel, April 23, 2015. Date of retrieval: May 8, 2018.
- [13] <http://www.dbms2.com>, DBMS2, *When it's still best to use a relational DBMS*, Monash Research Team, May 29, 2011. <http://www.dbms2.com/2011/05/29/when-to-use-relational-database-management-system/>. Date of retrieval: May 15, 2018.
- [14] Wikipedia, <https://en.wikipedia.org>, *Create, read, update and delete*. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete. April 17, 2018.
- [15] Wikipedia, <https://en.wikipedia.org>, *Database normalization*. https://en.wikipedia.org/wiki/Database_normalization. April 20, 2018.