NoSQL

Unai Perez Mendizabal

January 31, 2017

Abstract

The globalization and normalization of the use of the Internet, and with it all the services that have grows throughout these last years, has caused an exponential growth of data. This data, generated by plain use of the web services, such as just taking a look at a certain Amazon page or clicking a Google result, has become what is called Big Data: Massive amounts of data that can be used to find patterns, and to get to know the user and its needs. All this information has been so far stored in relational databases. But the increasing amount of data leads to a drop in their performance, causing enterprises to look away for new software that is ready to take care of this big workload. This would cause the raise of a new concept in database stores, what is being called as NoSQL. The technologies that comprehend NoSQL take a different approach in how the information is structured and stored. The objective of this paper is to analyze what different types of NoSQL systems exist, and how they differ from both other NoSQL types and relational systems, in their implementation and in their ideal use scenarios, and how suitable they are for Big Data applications.

Contents

1 RDBMS vs. NoSQL: Introduction **1**

2 About CAP theorem and BASE **1**

3 NoSQL database types **2**
3.1 Key-value Store Databases (KVDBS) . . . 3
3.1.1 Redis . . . 4
3.1.2 Riak KV . . . 6
3.2 Graph Databases (GDBS) . . . 7
3.2.1 Neo4j . . . 9
3.3 Document-oriented Databases (DODBS) . . . 11
3.3.1 MongoDB . . . 12
3.3.2 Amazon DynamoDB . . . 14
3.4 Column-oriented Databases (CODBMS) . . . 17
3.4.1 HBase . . . 18
3.4.2 Apache Cassandra . . . 21
3.5 Search engines . . . 23
3.5.1 Elasticsearch . . . 25
3.5.2 Apache Solr . . . 27

4 Scalability in NoSQL **30**

5 What and when to use **31**

6 Live testing **33**

7 Conclusions **34**

References **36**

Appendices **37**

A Getting recommendations from Neo4j **37**

B CRUD operations in DynamoDB **37**

C Setting HBase up **40**

D Using HBase shell **40**

E HBase Java API examples **41**

F CRUD operations in MongoDB **44**

List of Figures

1 Simple relationshipship example, graph databases 8
2 How a table is stored row by row 18
3 How a table is stored column by column 18
4 Example result with various versions, HBase shell 20
5 Creating a record in index bank, type account, Elasticsearch . . . 26
6 Search with query string parameters (in URL), Elasticsearch . . 26
7 Search with request body parameters, Elasticsearch 27
8 Posting documents, Solr 28
9 Getting all documents, Solr 28
10 Getting documents under conditions, Solr 29
11 Excluding words from the search, Solr 29
12 Simple configuration, HBase 41

List of Tables

1 Table with example values 17
2 Representation of a table with two column families, HBase 19
3 Quick comparison of NoSQL types 35

List of source codes

1 Querying actors that worked with directors of Cloud Atlas, SQL 10
2 Querying actors that worked with directors of Cloud Atlas, Cypher 10
3 Retrieving all data in collection, MongoDB 13
4 Querying all people called James, MongoDB 13
5 Querying all people called James older than 30, MongoDB 13
6 Defining primary key types, DynamoDB 15
7 Defining primary key's field type, DynamoDB 16
8 Defining the table's throughput, DynamoDB 16
9 Putting table creation parameters to work, DynamoDB 16
10 Creating and filling a table, HBase shell 20
11 Allowing and getting versions, HBase shell 20
12 Creating a keyspace and a table, CQL 22
13 CRUD operations, CQL 23
14 Finding common co-actors for T. Hanks and T. Cruise, Neo4j . . 37
15 Example dataset for Steam, Neo4j 38
16 Getting game suggestions, Neo4j 38
17 Example table, DynamoDB 39
18 Putting data, DynamoDB 39
19 Getting data, DynamoDB 40
20 Creating a table, HBase shell 41
21 Enable, disable and delete tables, HBase shell 41
22 Putting, getting and scanning a table, HBase shell 42
23 Creating a connection with Java API, HBase 43
24 Creating a table with Java API, HBase 43
25 Disabling and deleting a table with Java API, HBase 43
26 Putting and getting data with Java API, HBase 44
27 Inserting a restaurant, then getting all, MongoDB 45
28 Conditioned select, MongoDB 45
29 Using OR and AND, MongoDB 45
30 Updating and deleting documents, MongoDB 46

1 RDBMS vs. NoSQL: Introduction

The classic data structure in Relational Database Management Systems (RDBMS) has never changed, since it was postulated in "A relational model of data" [1] by Edgar F. Codd , an IBM engineer, in 1970.

Data, identified by an unique primary key, is stored in tables that represent a specific type of information. Each table has a set of columns that represents different properties of the said type of information. Tables are related to each other implicitly with foreign keys, this is, columns in tables that reference primary keys on other tables.

The name of relational databases comes from the relationship between the tables. The system is strict: Once a relationship between two or more tables is defined, it must be satisfied. This is, the foreign key columns must include a primary key that exists on the referenced table, though sometimes a null foreign key is used to reference no relationship at all.

With the growth of Big Data, the need of developing different approaches to store and retrieve data has also grown. This amount of information needs a more flexible database that handles the data in different manners, with different concepts and techniques. There may be or may not be a relationship between records. The object-like row structure may be gone as well.

NoSQL systems aim to change the way data is stored and structured. The name comes from "Not Only SQL", rather than from "No SQL". The concept does not reject SQL, just the relational structure of the data. Instead, they come up with new points of view on how to store information, whether they use SQL or another type of language.

In the next pages, different approaches of non-relational databases are going to be explained and discussed. Both relational and non-relational systems have their advantages and disadvantages, which will also be talked about later.

2 About CAP theorem and BASE

Also named Brewer's theorem, as explained in [2, 3], it states that it's impossible for a distributed computer system to provide in the same time the following guarantees:

- **Consistency**, what means every read must receive the most recent write.
- **Availability**, which means that every request must receive a response, without guarantee that it contains the most recent version of the data.

- **Partition tolerance**, which means that the system can continue working when something prevents communication between nodes to happen correctly.

It's clear which of the guarantees is what RDBMS don't fit, and that is one of the reasons of the rise of NoSQL systems. This is partition tolerance. Classic RDBMS have trouble dealing with network partitions because replication between the master and the slave is critical, and a partition would cause a much larger failure.

With big data nowadays, partitioning has become critical, as a single node system would not be able to handle all the load that is generated. That's the big problem with RDBMSs like Postgres or MySQL.

Some of the NoSQL systems have loosened up the requirements of consistency to achieve better availability and partitioning. This created the concept of *Basically Available, Soft-state, Eventually consistent* or **BASE**[1] systems, as explained by Pritchett in [4], which comes in contrast with *Availability, Consistency, Isolation, Durability* or **ACID**.

The meaning of BASE is, in more detail, as follows:

- The system is **available**, this is, every request will have a response, but only **basically**, this meaning that the response could be an error.
- The system will **eventually** become **consistent**. Data will propagate to where it should sooner or later, but input will still be received, even though the system is not checking the consistency of all transactions.
- Lastly, systems also are **soft-state**, meaning that even when there is no input, changes may still be ocurring due to the eventual consistency, thus being the state always soft.

Data with NoSQL databases is treated differently, that's why a concept like BASE was needed to define how the systems work. While these still apply to BASE, the CAP theorem still applies as well. Some of the weaknesses of the later on discussed systems are based on this theorem, and will in the pertinent moment be explained.

3 NoSQL database types

Wikipedia's entry on NoSQL databases lists up to 8 different types while DB-Engines.com lists 8 as well, but with some differences (They both exclude the multi-model concept, though). It's clear that different sources make up

[1] ACID vs. BASE: `http://bit.ly/2gskONO`

their own interpretation of the current classification of NoSQL systems. Still, the main types are considered to be key-value DBs, document-oriented DBs, graph DBs and column-oriented DBs. Those are going to be discussed from now on.

Also, some consider search engines as some kind of NoSQL store. Software like Elasticsearch and Apache Solr are getting more popular than some of the systems discussed below. If they are databases or not, that is up to interpretation. But considering the strength they are gaining, it's sensible to discuss them too. So, they are going to be talked about later on as well.

3.1 Key-value Store Databases (KVDBS)

They are based on the concept of associative array, also known as dictionary (like in Python) or map (like in Java). The database stores records or value-pairs that are formed by the key and the value. The unique key is used to identify the value assigned to it.

The value is stored as a blob (Binary Large OBject) so technically it can be anything: An image, a file, an object (as a set of fields) It is opaque for the system. That, at the same moment, makes it difficult to know what type of value to expect from a request, and it is up to the developers to figure out how to handle the information going to and coming from the database.

As mentioned, this type of database treats all kinds of data equally, as just an object. This property makes the KVDBS really flexible. Objects may have optional fields that in some cases are not going to be defined. With the KVDBS these null values need not to be stored, leading to less memory use and better performance.

In general, KVDBS do not use any kind of query language. They provide simple commands, such as get, put, and delete to retrieve, insert and delete records, as a map would in Java. The simplicity of this approach makes these systems easy, portable and scalable.

Coming to talk about CAP theorem, different implementations have different problems. For example, Redis and Aerospike are CP, this is, they provide consistency and partition tolerance. These systems usually have problems with availability while keeping data consistent across partitions. Amazon's dynamo and Riak, on the other hand, are AP, meaning they provide availability and partition tolerance. What this causes is called "eventual consistency", with which we know data will be consistent but not at all times.

Software like memcached and Hazelcast, the approaches of which sound familiar to a regular KVDBS, openly state that they are not a database system. Memcached is described in its homepage as an in-memory key-value store for small chunks of data from results of database calls, API calls, or page rendering. This is, it works as a cache that gives the result of a request, preventing the actual application from accessing costly external sources. It works following the same principle as KVDBS, though. This approach is called caching, an explanation of which, applied to memcached, can be found in [5].

Different types of KVDBS come from their implementation. Wikipedia entry "Key-value databases" and Aerospikes "What is a key value store" page refer to these types. One of the types is defined by whether or not the system supports key ordering. BerkeleyDB and HyperDex are some examples that do support ordering. Some KVDBS differ in where they store the data, which may be either RAM (e.g. Hazelcast and Aerospike), SSD and/or rotating disks (e.g. Couchbase) or a combination of more than one. Then again, Hazelcast[2], in its homepage, differences the systems that use RAM and the ones that use disk, saying that systems as Hazelcast itself, which use RAM storage, are not actual database systems, referring to the aforementioned caching.

The main advantage of RAM storage systems is the processing speed. Its much easier and faster to access and/or write in memory than it is to write in a primary disk. In general, though, the available RAM storage is significantly smaller than the available disk space. Its cheaper to buy more disk than it is to buy more RAM memory. Also, RAM is volatile, therefore, all data will get lost if any unexpected crash or some kind of fail happens.

Now, two different implementations will be explained in detail. Based on the KVDBS ranking of popularity[3], Redis, the most popular in-memory based key-value store, and Riak KV, the most popular on-disk based key-value database, have been chosen.

3.1.1 Redis

As it reads on its frontpage, Redis is an open source (BSD license), in-memory data structure store. Its developed by Italian developer Salvatore Sanfilippo. The names original meaning is Remote Dictionary Server. Development started on early 2009. It earned popularity, so VMWare hired Sanfilippo to work full-time on Redis.

It's easy to use and simple. There is no query language, only commands, which depend on the data type that is to be handled. The data types[4] are as follow:

[2]Hazelcast.org, in-memory NoSQL: `http://bit.ly/2gGfV6t`
[3]DB-Engines Ranking of key-value stores: `http://bit.ly/2fZgmtq`
[4]Redis data types: `http://bit.ly/2gsn2wJ`

- Binary-safe **strings**
- **Lists**, which are a collection of strings in order of insertion.
- **Sets**, lists of unique unsorted strings.
- **Sorted sets**, which are like regular sets, only each element has a score that determines its position in the set. This score is set on insertion.
- **Hashes**, maps in which a key string has a value string. Key-value pair list.

Also: Bit arrays, which provide a way to handle strings as an array of bits, and HyperLogLogs[5], probabilistic data structures used to count unique elements in sets.

Regarding HyperLogLogs, calculating the cardinality of a set with precision requires an amount of memory proportional to the number of unique values. That is because there is no way to know if a value is repeated but comparing it to all previous values. HyperLogLogs solve this problem by trading memory consumption for precision. There is a standard error of 2% to estimate cardinalities larger than 10^9, but memory usage is only 1.5 kilobytes. This is a unique feature of Redis. It can be used, for instance, to know how many unique users have visited the web page today.

Redis also services a working implementation of the publish/subscribe messaging paradigm[6]. With this service it is possible for a client to publish a message to a certain channel, and all the clients that are subscribed to that channel will receive the message instantly. Any client can subscribe to or unsubscribe from any channel.

Installing Download the last stable version from the Redis' frontpage and unzip. Compile the source code by running the `make` command (using msys on Windows) to generate the `redis-server` and `redis-cli` executables. Then, it is recommended to add the directory of the executables to `PATH`.

Running Just launch the `redis-server` executable from the command line or terminal and pass the path to the configuration file as the argument (the default file is in the parent folder of `redis-server`, but still the path is needed as an argument), then keep it open.

[5] Sanfilippo on Redis HyperLogLogs: `http://bit.ly/2fZbR24`
[6] About pub/sub pattern on Wikipedia: `http://bit.ly/2gaEOGS`

Using `redis-cli` is an out-of-the-box client, which can be used to directly issue commands to the server. It's also possible to access the database via the APIs that are available for different languages[7], from better-known Java to less-known R. Some frameworks, such as Spring[8], simplify the work by adding high-level access to the database, so that coping with the connection and its configuration is no longer needed.

Testing Two example back-end REST APIs have been written to test Redis, one using the Jedis Java Redis driver and the other using Spring Data Redis framework. More thorough information on the tests will be given in the live testing section.

3.1.2 Riak KV

It's the key-value database implementation of company Basho. They describe it as a distributed NoSQL KVDBS with advanced local and multi-cluster replication that guarantees reads and writes even if hardware or network failures occur. There are five versions of the product, with varying pricing and services, including an open-source version[9] the first commit of which was back on July 2010. It is written in Erlang. Its strength is how easy it is to create a cluster with different instances of Riak server, this is, nodes.

It has many APIs in different languages to support client development. In particular, Basho offers support in a range of languages that go from Java to Erlang, and also has community supported libraries, including C and Perl. Though it does not include a client out-of-the-box, all operations can be made via HTTP request thanks to the HTTP API[10]. Riak is supported on platforms such as Amazon Web Services or Windows Azure, and works on Ubuntu and Mac OSX, among others.

Buckets, in Riak KV, are the namespace in which the information is stored. With buckets, it's possible to use the same key to store different information, in different buckets. It's comparable to a folder in a filesystem, or, more vaguely, to a table in a RDBMS. Each bucket belongs to a bucket type, which allows it to have a default specific configuration. Amount of copies of object per cluster, or prevalence of writes can be configured using bucket types. All of this is optional, if no bucket is specified the default will be applied.

Riak supports the following data types:

- **Flags** are like boolean values, except that instead of true or false, flags' possible values are `enable` or `disable`. Can't be used on their own.

[7] Redis clients: `http://bit.ly/2gGjvh2`
[8] Spring Data Redis: `http://bit.ly/2gXQxK4`
[9] Riak KV on Github: `http://bit.ly/2gGiIfW`
[10] Riak KV HTTP API Docs: `http://bit.ly/2fZfrcK`

- **Registers** are the equivalent of Redis binary strings. Can't be used on their own.
- A **counter** stores an integer that can be incremented or decremented, and can be used as an object associated to a bucket/key pair or within a map.
- **Sets** are collections of unique binary values (as strings). Like counters, they can be associated with a bucket/key pair, or used within a map.
- Finally, **maps** are a combination of all other data types that form something comparable to an object. Flags and registers can only exist within a map, and maps can embed maps as well.

Installing Download the installer from the web page[11]. .deb and .rpm packages (among others) are provided, but the source is also available to compile and install manually.

Running If Riak was installed from package, the `riak` command will be available from terminal. Issuing just `riak` will show all available options. `riak start` will start the server in background, `riak console` will start the server on the console, printing all logs on screen. `riak stop` will stop the background server.

Using Before running the server, some configuration is required. In the `/etc/riak/riak.conf` file, change the listeners (`listener.http.internal` for HTTP and `listener.protobuf.internal` for Google's protocol buffers) for them to bind to the correct IP addresses. Clients will have to use these listener addresses, so it's important to check they are correctly configured. The rest of the settings are optional, but it is encouraged to check them for sanity. Once the server is running, data can be managed directly through the HTTP API or by using one of the clients.

3.2 Graph Databases (GDBS)

The concept of data changes radically from the relationshipal database to these systems. In RDBMS, tables are what form the relationships, this is, information in one table must always have a relationship with some column in some other specific table; whereas in graph databases, the data itself forms relationships with other data, as analyzed by Angles and Gutierrez in [6].

Data in graph databases[12] is presented using nodes, which can have properties (key-value pairs) and can be tagged with labels that describe what kind of information they store. Nodes are connected using relationships, called edges, which also have tags to describe what kind of relationship joins both nodes. The later ones also can have some properties related to the relationship itself.

[11] Download Riak KV: `http://bit.ly/2g1LLGQ`
[12] Neo4j.com, Graph Databases: `http://bit.ly/2gT5Hxg`

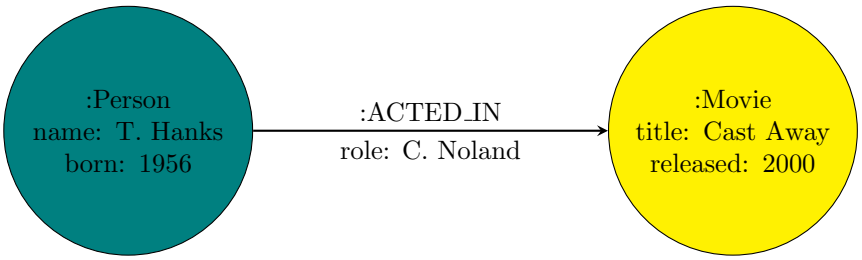

Figure 1: Simple relationshipship example, graph databases

As seen in the figure 1, there are two nodes, which correspond to actor Tom Hanks and movie Cast Away. Both nodes have two properties. The node in the left has *name* and *born*, and the right one has *title* and *released*. One has the *Person* label and the other has *Movie*. Their goal is to help describe what type of information the nodes store, and they don't add any constraint (e.g. don't make any property mandatory). The arrow, which represents an edge, means that the nodes are related. The edges also have properties and labels that describe the relationship. In this case, it's possible to know that the relationship between Tom Hanks and Cast Away, is that the actor acted as C. Noland in the movie.

It's also important to note that the direction in which the edge points is critical, because it's not the same to say (and store) that Tom Hanks acted in Cast Away or that Cast Away acted in Tom Hanks. There is a difference between **ingoing** and **outgoing** edges.

The strength of graph databases is the concept of data itself having some relationship with other data, and also how visually they are able to display the information. It's possible to modify just the relationship between two nodes without having to modify any property of any of them. Also, it's impossible to delete a node if it has any relationships unless explicitly told to, kind of like with RDBMS foreign keys and their "on cascade" configurations.

Each node in a graph database has one set for ingoing and another for outgoing edges. What in RDBMS can be done using joins, which are rather costly and take some time to make (especially when there is a lot of data stored), can be done at very little cost with graph databases, by accessing the mentioned sets directly. Moreover, the resulting data is simpler but more expressive than with regular joins.

The only weakness of the graph databases comes from the CAP theorem. Emil Eifrem, CEO of Neo4j, answered the question of how CAP theorem applies

to Neo4j at a webinar[13] on July 13th, 2011. He answered that Neo4j is not partition tolerant. As of today, sharding and scaling still is the biggest problem of any GDBS. To run multiple nodes with a graphic database would require replication of all data in a master-slave fashion, making the cluster dependant on the network.

Another minor problem is the lack of standard query method. Learning to use a RDBMS requires to learn SQL, and most if not all RDBMSs use SQL with some minor implementation differences between them. Neo4j uses proprietary Cypher language, OrientDB uses a variation of SQL, other use HTTP APIs or Java libraries, and so on.

The ranking of most popular graph databases[14] shows that Neo4j has absolut control of the GDBS market. On the following pages, Neo4j, the most popular one by far, will be discussed in detail to learn about how to use these type of systems.

3.2.1 Neo4j

Developed by Neo Technology, Neo4j, the current most popular graph database, was initially released in February 2010. It is written in Java and Scala. It has an open source community edition. In December 13th, 2016, major update 3.1 Enterprise version was released. The company also has also released a Java library.

Neo4j organizes data just as was explained previously. There are nodes, with a label and some properties, that are related to other nodes via edges, which may also have label and properties. Edges go from one node to another, this is, are outgoing from one node, and ingoing to another node. There is no primary key in the old-fashioned sense. Properties are key-value pairs in which the key must be a string and the value can be any primitive type (numeric and strings).

Even though there is no primary key, an ID field is used to identify records internally. But it is discouraged to use them, as Neo4j itself sets and uses them, and values may change version to version. The different properties can be used to search for a specific record. Or it is possible too to create a constraint on a property name for the values of it to be unique, this way the primary key functionality can be mimicked.

The most notable features of Neo4j are the Cypher language, the true strength of this system, and the HTTP Rest API and browser-based client that can be used to send Cypher queries to the server. The Rest API is normally used

[13] Neo4j Webinar Series #1: `http://bit.ly/2gXW8QQ`
[14] DB-Engines ranking of graph databases: `http://bit.ly/2gY4QhF`

to access the database and the information from languages that are not directly supported. The browser app is a good way to learn and use the database in a sandbox mode, to make data sanity checks, and also to see data graphically.

Using the movie database example provided by Neo4j, here is a comparison of how a complex query can be written in SQL and the equivalent Cypher code. Say the goal is to get the names of all actors that ever worked with the directors of "Cloud Atlas" (even in other movies). Code on listing 1 would do the job in SQL, and code on 2 in Cypher.

```
SELECT DISTINCT actors.name AS Actor FROM Person actors
INNER JOIN ActedInMovie acted
    ON acted.actor_id = actors.id
INNER JOIN DirectedMovie directed
    ON directed.movie_id = acted.movie_id
INNER JOIN Movie movies
    ON movies.id = directed.movie_id
WHERE movies.title = "Cloud Atlas"
```

Listing 1: Querying actors that worked with directors of Cloud Atlas, SQL

```
MATCH (actors:Person)-[:ACTED_IN]->(:Movie)
<-[:DIRECTED]-(:Person)
-[:DIRECTED]->(:Movie {title: "Cloud Atlas"})
RETURN DISTINCT actors.name AS Actor;
```

Listing 2: Querying actors that worked with directors of Cloud Atlas, Cypher

Note that, for listing 1, it was assumed that the relationships between actors and movies are stored in a table called `ActedInMovie` with columns `actor_id` and `movie_id`, and that relationships between directors and movies are stored in table `DirectedMovie` with columns `director_id` and `movie_id`.

As seen in the examples above, in the relational SQL code three joins are required, which makes the query computationally expensive. With Cypher instead, the query is much faster run because the access to all relationships is much faster and straight forward, and it's much more visual and easier to understand as well.

Another feature that would required of a lot of power and would be costly to do in a relational database (because many joins would be required), is suggestion queries. It's not a specific feature, it's rather a way in which Cypher queries can be done, such that they can make suggestions. It's based in

relationship chains. To put a simple example, if the database knows that person A likes things X and Y, and if person B likes X, it's likely for person B to like Y as well. More thorough examples can be found in appendix A

Installing Installers are available to download on the website for Windows, Mac and Docker. For Linux it's possible to download a `.tar` with all needed files. Just unzip where wanted.

Running In Linux, using the `neo4j` file, it's possible to start and stop the server in the background, and it's possible to run the server on the console as well. In Windows, it is installed as a service.

Using As mentioned before, it's possible to communicate with the server through the HTTP Rest API or the browser app.

Testing Two different REST APIs have been written to test Neo4j. One uses the official Java driver and the other works with Spring Data Neo4j framework. More thorough information on the tests will be given in the live testing section.

3.3 Document-oriented Databases (DODBS)

In contrast to the strict relational concept, with DODBSs, each record and the information associated to it is thought of as a *document*. Everything related to any document is encapsulated together. While in a RDBMS a schema is to be designed before getting to use the database, in DODBSs, all the information is stored together into a document.

Basically, the document-oriented concept is a more developed subclass of key-value stores. The difference lies in the fact that document-oriented databases store the information in a structured format that the system itself can understand.

Each server may have one or more database. Databases have collections, in which documents are stored. Normally, documents in a collection represent one type of information, which makes collections vaguely comparable to a RDBMS table. Anyway, this is completely up to the developer, there is no constraint as to what structure the data must follow in a certain collection, nor there is any way to enforce one.

Quoting Couchbase's example[15], say there is a table with beer and another one with brewery information in a RDBMS. Beers are made by breweries. So, obviously, the beer table will have a foreign key column, pointing to the brewery table's primary key.

[15] Couchbase on DODBSs (with example):`http://bit.ly/2hob4Yh`

There are two options when converting the information in the example to documents. It's possible for either the brewery or the beer to contain all the information of the other on their document. Or else, a document can be created for each one of them and add a field in the beer's document to showcase an implicit relationship.

The goal of this example is to prove that there is no such thing as a schema or a structure. The documents can have any number of fields which can have any primitive kind of value, any structure that may or may not correlate to other documents in the same collection or database.

Storing data this way has some advantages. One is that documents are independent units, which improves performance, as all related data is read contiguously. Also, that makes it easier to replicate data while preserving locality. Another advantage is that application logic is easier to write, as the plain object in the code can be turned into a document. Lastly, migrations are not as costly, because no schema is needed.

Of course, DODBSs also have some drawbacks. Referring back to the CAP theorem, they are said to be CP (consistent and partition tolerant). Moreover, if reads from non-master nodes are permitted, the systems become eventually consistent, so reads might get out-of-date results. Also, representing some information in documents, especially coming to subclasses, may cause lots of duplicate information.

The two implementations of the document-oriented concept that will be explained below, are the two most popular solutions of the kind, according to DB-Engines' ranking[16].

3.3.1 MongoDB

Developed by the homonym company, it's an open source document-oriented database, first released on 2009. As of the writing of this document, its last stable version is 3.2.11. It's developed in C++. According to DB-Engines, it's the most popular of the NoSQL databases.

Instead of using tables, like in regular RDBMS, information is stored in a JSON-like format. Specifically, a JSON superset called *Binary JSON* or **BSON** is the standard interchange format of MongoDB. It is designed to be faster and more efficient, but it might take more space than JSON. It implements some data types that JSON does not have, like date or even Javascript code.

[16] DB-Engines Ranking of Document Stores: `http://bit.ly/2hEkah3`

As BSON is a superset of JSON, it supports primitive data types such as strings, numbers, booleans, null values, arrays and objects. In addition, it also supports the aforementioned date and Javascript code types, byte arrays, MD5 binary data and regular expressions. As BSON is the interchange format used by MongoDB, all these types are supported as well in the database.

Data in MongoDB is stored in collections, which, as explained before, are similar to tables in RDBMS: They usually store similar data but no constraint is imposed. These collections belong to databases, which have their own administration settings, like users and roles. Information in one collection, or even in one database, can point to information in another collection or database, by storing the id field, collection and database of the information in the document.

MongoDB provides a client with which data can be queried using BSON and Javascript. It supports database management commands that allow from showing the collections or the database to managing roles. For example, to query all information in a collection, check listing 3. Or, to find data that matches certain information, the BSON filter is used, as in listing 4.

```
db.collection.find().pretty();
```

Listing 3: Retrieving all data in collection, MongoDB

```
db.collection.find({name: "James"}).pretty();
```

Listing 4: Querying all people called James, MongoDB

Querying with more complex conditionals (like "greater than") requires operators (BSON keys starting with $). These are BSON filters with a specific structure. For example, to find any person named James that is older than 30, query in listing 5 is used.

```
db.collection.find({name: "James", age: {$gt: 30}}).pretty();
```

Listing 5: Querying all people called James older than 30, MongoDB

MongoDB's query language is completely different from SQL, but it's very logical, and robust once the developer gets used to it. Operators give all kind of functionality to the BSON filters and are proof of how a format so closely related to JSON can have more complex functionalities.

To see more examples on how to perform *CRUD* operations on MongoDB, refer to appendix F.

Installing Downloads are available for Linux, Mac and Windows Server on the website's download center[17]. The Windows download is an installer that will do all the job. The Linux version is a `.tar.gz` that can be extracted to the preferred directory.

Running To start the server in Linux, just run `mongod` on the extracted folder's `bin` directory. In Windows, similarly, run the `mongod.exe` executable in the installation directory.

Using It's possible to manage and query the database directly with the out-of-the-box client `mongo` in Linux and `mongo.exe` in Windows. Clients with UI are available from third parties, like MongoClient[18]. It's possible to connect via the libraries for different languages.

Testing Two different REST APIs have been written to test MongoDB. One uses the official Java driver and the other works with Spring Data MongoDB framework. More thorough information on the tests will be given in the live testing section.

3.3.2 Amazon DynamoDB

Created by Amazon, it was initially released in 2012. It is based on the Dynamo principles of storage. DynamoDB is an strictly commercial software, which means that it is necessary to buy it to use it. A free instance of the server can be used locally for development, though.

The most noteworthy feature of DynamoDB is that it is purchased by specific throughput rather than by storage, that can be set per table at any time. In Amazon's words, this is called predictable performance. All the scaling, redundancy and such techniques are automatically made by Amazon itself.

Even though it clashes with the NoSQL principles, DynamoDB is a document-oriented store with relationships between tables, like in RDBMS. There is no schema in the old-fashioned way. Primary keys and foreign keys have to be specified on table creation time. But the records inserted in the tables do not follow any specification, columns don't exist. Any key-value pair can be inserted within a document, which must be identified by a primary key, the only constraint.

[17] MongoDB Download Center: `http://bit.ly/2hBcpv9`
[18] MongoClient: `http://www.mongoclient.com/`

There are two types of primary keys depending on the key type and three depending on the attribute type. The attribute type just means that the primary key can be either a string, a number or a binary. The key type means that the primary key can be a partition type ("hash") or a sorting key ("range").

This must be explained in more detail: records can be identified by a simple primary key, formed by one partition key, or by a composite key, formed by one partition key and one sorting key. When storing data, DynamoDB divides a table's items into multiple partitions, and distributes the data based on the partition key value. With a composite key, many values can be stored in the same partition, then sorted with the sorting key.

DynamoDB offers a JavaScript shell (both locally or when purchased online) via which all operations can be made in a clear JavaScript code. When executed locally, it's possible to access it with a browser via `localhost` on port 8000, in path `shell`. It provides a console to issue JS commands directly, and a text editor with which a bunch of code can be directly issued to the database, instead of just one command.

As example, say the goal is to create a table with a string partition key called `Date` that represents the date of sale of one item, represented with number `ItemId`. First, the parameters of the table creation must be defined, starting with defining the key types and names, as shown by listing 6.

```
var params = {
    TableName: 'MyTable',
    KeySchema: [ {
        AttributeName: 'Date',
        KeyType: 'HASH'
    }, {
        AttributeName: 'ItemId',
        KeyType: 'RANGE'
    } ],
    ...
};
```

Listing 6: Defining primary key types, DynamoDB

Then, the attribute type of each of the primary keys just defined must be defined as well. They must be either 'S' for string, 'N' for number or 'B' for binary, as seen in listing 7. The last thing that must be defined is the throughput of the table, as explained previously and shown by listing 8.

```
var params = {
    ...
    AttributeDefinitions: [ {
        AttributeName: 'Date',
        AttributeType: 'S'
    }, {
        AttributeName: 'ItemId',
        AttributeType: 'N'
    } ],
    ...
};
```

Listing 7: Defining primary key's field type, DynamoDB

```
var params = {
    ...
    ProvisionedThroughput: {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
};
```

Listing 8: Defining the table's throughput, DynamoDB

Last step is to pass the defined settings to the API's table creation function. When the function ends, it calls the fallback function that must be set. If all worked correctly, the `err` argument will be null. This is how the responses are handled for all requests in DynamoDB, with fallback functions. See example in listing 9.

```
dynamodb.createTable(params, function(err, data) {
    if (err) {
        //handle error somehow
    } else {
        //do something
    }
});
```

Listing 9: Putting table creation parameters to work, DynamoDB

Complete examples on how to insert and retrieve data can be found in appendix B.

Installing Definitely, DynamoDB is different from the other discussed systems. It's possible to purchase it from its website[19], or download[20] it to develop locally. No need to install, just extract in the preferred directory.

Running For local use, run the following command on the terminal or command line:

```
java -Djava.library.path=. -jar DynamoDBLocal.jar
```

Using Access the console on `localhost:8000/shell` to use it directly or use one of the official available APIs for Java, PHP or .NET.

3.4 Column-oriented Databases (CODBMS)

Even though the approach in which data is interpreted and visualized resembles the RDBMS general approach, the difference stays beneath. CODBMSs use tables to structure information of similar type, but schema-less, this is, with no need to design the structure of the database before-hand. The table has records, as in RDBMS, identified by one id. But each record may have the columns that it's needed for it to have, sort of like documents do in DODBSs.

The big difference just mentioned lays in how the data is persisted, as Abadi et. al. explain in [7]. Take for example, the RDBMSs, which are row-oriented. The information is stored in the disk row by row, meaning that all information related to one record in a row will be stored sequentally. As opposed to that, CODBMSs store all the information of one same column sequentally.

To show it more graphically, say the table 1 is in the database. `RowId` shall be an internal identification value for the row.

RowId	EmpId	FirstName	LastName	Salary
001	10	Joe	Smith	40000
002	11	Mary	Jones	50000
003	42	Cathy	Johnson	44000
004	57	Bob	Jones	55000

Table 1: Table with example values

Now, data in that table will be persisted on disk as shown in listing 2 in a row-oriented system. But in a column-oriented database though, it will be stored as shown in figure 3.

[19] AWS DynamoDB: `http://amzn.to/2i4IPyV`
[20] Download local DynamoDB: `http://amzn.to/2iawOE2`

```
001:10,Joe,Smith,40000;
002:11,Mary,Jones,50000;
003:42,Cathy,Johnson,44000;
004:57,Bob,Jones,55000;
```

Figure 2: How a table is stored row by row

```
10:001;11:002;42:003;57:004;
Joe:001;Mary:002;Cathy:003;Bob:004;
Smith:001;Jones:002,004;Johnson:003;
40000:001;50000:002;44000:003;55000:004;
```

Figure 3: How a table is stored column by column

It is worth noting that if a value is repeated in one column, instead of storing the value many times, the value is stored once, but referencing all the rows that contain it. This can be seen in the example above, with value `Jones`, which is in the third column. These examples are not to be taken strictly, it's just a simplification of the actual way the databases work.

The way column-oriented databases store data reduces the amount of data to be read from disk, because it compresses the columnar data and by reading only the data necessary to answer the query. Writes are more complex to do, though, and thus slower. Inserts have to be split into columns and then be compressed when storing.

3.4.1 HBase

It's an open source system that is developed by Apache and is part of the foundation's Hadoop project. It's written in Java. Its first stable release (version 1.0) was on the first quarter of 2015, after being in development 7 years.

Apache Hadoop is a framework that allows the management of very large data through clusters of computers, each offering local computation and storage. Put it simply, it allows to store files bigger than the disk of the computer, and shares the computation of all computers in a cluster. It does this using its own file system implementation, the Hadoop Distributed File System (HDFS), on top of which works HBase. Cassandra is part of Hadoop framework as well.

As L. George explains in [8], HBase organizes the data in tables that do not have any relationship with other tables. They are formed with column families. Those are groups of columns that store related information. Column families must be defined when creating the table, but can be modified afterwards.

Each record is identified by a unique key that must be defined when inserting for the first time in the said row. The most common practice is to give it a unique string that relates to the information inserted (refer to the later example for a clearer glimpse).

Each row then can have any column that is required inside the already existing column families. These work in a similar fashion as documents: they are a collection of key-value pairs, always related to the column family.

Another interesting feature is that HBase is able to remember past values of columns if told to. When creating a table (or altering it later on) it's possible to set the `version` parameter to a number. This is the number of changes the table will remember. Then, when getting the information from the database, it's also possible to ask for more than one version, to which the database will respond with as many past value as asked, with the timestamp of when they were inserted.

Say the goal is to store information about a person. The needed information is the address, divided by city, street and number, and the full name, divided by first name and last name. Translating this to HBase format, address and full name would be column families, the first formed by columns city, street and number, and the last formed by columns first name, and last name. See a representation in table 2.

	People				
	Address			Full name	
	City	Street	Number	First name	Last name
uperez	Ordizia	Majori	24	Unai	Perez M.

Table 2: Representation of a table with two column families, HBase

HBase provides a Java client API with which all operations can be executed. It also provides REST, Avro (a data serialization framework, also part of Hadoop) and Thrift (a binary communication protocol by Apache) gateway APIs.

To operate directly with the database, HBase comes with a shell. It automatically connects to the local server to perform the operations. It's possible to administrate the tables and perform CRUD operations with simple commands such as `put` or `get`.

For example, the following set of commands could be used to create and initialize a table following the graphic representation above.

```
create 'people', 'address', 'fullname'
put 'people', 'uperez', 'address:city', 'Ordizia'
put 'people', 'uperez', 'fullname:firstname', 'Unai'
...
```

Listing 10: Creating and filling a table, HBase shell

If it is required, for example, for the database to know the last five addresses of any person stored, the table should be created passing the `VERSIONS` parameter, as seen below. So, when getting the information from the database after a couple of changes, then getting it back as shown in listing 11, the result resemble what is show in figure 4.

```
create 'people', {NAME=>'address',VERSIONS=>5}, 'fullname'
// (Inserting some values) //
get 'people', 'uperez', {COLUMN => ['address:city',
'address:street', 'address:number'], VERSIONS=>5}
```

Listing 11: Allowing and getting versions, HBase shell

```
COLUMN CELL
address:city timestamp=1483386091220, value=Ordizia
address:number timestamp=1483386134480, value=24
address:number timestamp=1483386125491, value=43
address:number timestamp=1483386113730, value=12
address:street timestamp=1483386132092, value=Majori
address:street timestamp=1483386122005, value=Etxezarreta
address:street timestamp=1483386102246, value=Gipuzkoa
```

Figure 4: Example result with various versions, HBase shell

Notice how, in the returned message, there are different values with different timestamps for the same columns. The most recent value for number is `24` and for street, it's `Majori`.

To see more example on how to manage data via HBase shell, refer to appendix D. To see some Java API examples for HBase, check appendix E.

Installing Download[21] it from the website and extract it in a folder of your choice.

[21] Download HBase: `http://bit.ly/2iXNNKh`

Running Before being able to run it, it's needed to configure it. The most basic configuration required to run HBase can be checked out on the "Quick Start"[22] page of their website. A summary of this can be found on appendix C. Once everything's set up, `bin/start-hbase.sh` must be run from the installation folder.

Using Issue commands from the shell by running `bin/hbase shell` from the installation folder. The next most direct way is using the Java API. As mentioned, REST, Avro and Thrift interfaces are also available, but must be configured separately.

Testing Only one REST API has been written to test HBase. It uses the official Java driver provided by Apache. More thorough information on the tests will be given in the live testing section.

3.4.2 Apache Cassandra

It's a distributed database management system, free and open-source. It's written in Java. It was first developed by A. Lakshman (co-creator of DynamoDB) and P. Malik at Facebook. In March 2009 it became an Apache project, and a year later Cassandra got its first release version.

Its main goal, aside from storing data, is to provide high availability with no single point of failure[23]. A group of researches from University of Toronto that studied NoSQL systems stated that in terms of scalability, the clear winner is Cassandra, as shown in [9]. It is usually classified as AP following the CAP theorem, as availability and partition tolerance prevail over consistency.

Data in Cassandra is divided in keyspaces, which resemble the relational concept of databases. When creating a keyspace in Cassandra, its replication settings must be specified. These are the replication strategy and the factor. Tables then belong to keyspaces. Tables are sets of predefined columns, like in relational model. They must have a primary key as well.

Primary keys in Cassandra work differently from their relational counterparts. They are composed by two parts, the partition key, mandatory, and the clustering columns, which are optional:

- The **partition key** can be composed by one or more columns. Rows that share same partition key combination are considered a partition. So, in tables where partition keys are only one column, each row is a partition. All rows belonging to the same partition are guaranteed to be stored on the same set of replica nodes.

[22] HBase Quick Start: `http://bit.ly/2iXUXOx`
[23] Single point of failure: `http://bit.ly/2jLXEDi`

- The **clustering columns** define the clustering order for the partition of that table. In other words, they define the order in which the rows are stored internally. This makes the retrieval of ranges of rows within a partition very efficient.

All data in Cassandra is managed using a custom variation of SQL called CQL, which stands for *Cassandra Query Language*. Generally, SQL keywords and clauses keep their form and meaning, this is, to retrieve information `SELECT` is used, and to put some data `INSERT` is still used as well.

Due to it using a superset of SQL, it supports the same data types of SQL, as well as data types such as `inet`, which stores an IP address, or `uuid`, which stores a UUID[24].

In the examples that follow, some simple CQL statements will be shown and explained. For instance, see listing 12 on how to create a keyspace and a table in that keyspace. See listing 13 on how to select and insert data into Cassandra. Or refer to the CQL documentation at Cassandra's website[25].

```
CREATE KEYSPACE Internet
WITH replication = {'class': 'SimpleStrategy',
    'replication_factor': 1};
USE Internet;
CREATE TABLE Users (
    lastname text,
    firstname text,
    age int,
    registration_date timestamp,
    PRIMARY KEY ((lastname, age), firstname)
);
```

Listing 12: Creating a keyspace and a table, CQL

In listing 12, a table called Users, with partition keys `lastname` and `age`, and clustering column `fullname`, in keyspace Internet is created. This means that rows with the same `lastname` and `age` value will form a partition. Then inside the partition, rows will be ordered depending on the `firstname` value.

On the other hand, in listing 13, it's impossible to tell the difference between SQL and CQL. The syntax and the purpose of each statement is clear, it does not change. First, a new user is inserted. Then all users' `firstname`

[24] UUID: `http://bit.ly/1Pb5EbX`
[25] CQL documentation: `http://bit.ly/2iyZEAQ`

```
INSERT INTO Users (lastname, firstname, age, registration_date)
VALUES ('Perez', 'Unai', 21, dateof(now()));
SELECT firstname, lastname FROM Users
WHERE registration_date > '2015-01-01';
UPDATE Users
SET age = age + 1;
DELETE FROM Users
WHERE age < 18;
```

Listing 13: CRUD operations, CQL

and `lastname` is selected. Then, all users' age is incremented. And finally, all underaged users are deleted.

For someone that has experience in SQL, making the change to Cassandra would not be too tough. There are many different options that have not been mentioned so far, like `BATCH`, which works in a similar way to SQL transactions, or `TTL`, which lets specify the "time to live" in `UPDATE` and `INSERT` statements.

Installing Download Cassandra from the its webpage[26] and extract it where wanted.

Running In the `bin` folder of the extraction directory, issue the command `cassandra` to run Cassandra in the background, or `cassandra -f` to keep it attached to the terminal.

Using Data is accessible through the Java API or through the client executable, `cqlsh`, in the extraction directory's `bin` folder. It's used like MySQL's or MongoDB's client, by directly issuing CQL and receiving immediate response.

3.5 Search engines

They are an information retrieval system that search for specified keywords and returns a list of the documents (the *hits*) where the keywords were found. The concept has reached plenty of popularity due to one of the most used forms of search engines, this is, Web search engines such as Google or Bing.

The concept was first used when Emanuel Goldberg created a machine that could read binary data from physical cards with holes. Each card represented a book, so a card was used to search for metadata of one specific book. This was in 1931, as you can see in the patent [10].

[26] Download Cassandra: `http://cassandra.apache.org/download/`

Though not exactly databases, search engines do store information, up to petabytes of data. The stored information, instead of being records or objects, are documents that have keywords, which then are used to search. The key element is the way the information is retrieved.

Many search engine types exist, depending as well on the classification criteria used. Those are source, content type, and topic.

- They are classified by what **source** they search. For example, the source may be the Web (like with the examples mentioned previously) or the desktop (searching for a file in the computer), among others.
- They are classified by the **content type** they search for. They may search for full text, video or images.
- The **topic** they search for is also used as a criteria. For example, it's possible to search for bibliography or for medical literature.

Web search engines, or particularly Google in the last few years, have become the strongest search engines out there. But full text search engines are also gaining popularity and working out of the spotlight in many popular websites, such as GitHub, Facebook and Neflix.

In a full-text search, the engine examines all of the words in every stored document to try and find something that matches the criteria. When the number of documents to examine is not too high, the engine directly scans all the content of all documents with each query. This is called serial scanning. Unix command `grep` follows this strategy.

When the document number is large though, the task is divided into two tasks, indexing and searching. Indexing refers to the creation of an index in which all terms found in the documents, and a reference to the position it appears in the document. Then goes the searching process, which consists on searching in the index the wanted query.

The indexing part is done by the indexer. Depending on the indexer, it may ignore the so-called *stop words* like "the" or "and". Or it even may perform stemming, this is, reducing words to their root and using this to record them. For example, recording all conjugated forms of a verb as the infinitive of the verb.

As mentioned, search engines are not exactly databases, so comparing it with other database types or applying the CAP theorem is pointless. But it's important to mention two measures that are used to measure the quality of queries. One is recall, which is the amount of relevant results returned by a search over the actual amount of relevant results. The other is precision, which

is the amount of relevant results obtained with a query over the total amount of results.

Elasticsearch and Solr are the most popular of this subset of search engines. Movies and series in Netflix are searched using Solr, and articles on Wikipedia are searched for using Elasticsearch. They are indeed really used and so deserve to be discussed on the following.

3.5.1 Elasticsearch

An open-source, Lucene-based distributed full-text search engine, developed by Elastic. The development is led by Shay Banon, who previously created a precursor to Elasticsearch, called Compass, in 2004. When working on the third version of Compass, Banon realized that rewriting big amounts of code to achieve scalability would be needed. Thus, the first version of Elasticsearch was released in February 2010.

Data, in Elasticsearch, is distributed between **nodes**, this is, a running instance of the search engine, that may be in any of the machines available which form the **cluster**.

Then, data is also classified in **indexes**, that, hierarchically, are similar to databases in relational systems. Inside each index, data is divided by **type** or mapping. As the name says, all information usually in the same type, represent similar data. Types, also hierarchically, can be compared to tables in relational systems.

Ultimately, the stored data is called document or object. It is stored and represented in JSON, and, as it can be deduced, is stored in a type that belongs to an index. This information actually resembles documents in the fact that they can store any key-value pair, with no constraints. The main difference with Elasticsearch is that, like HBase, remembers the versions of each document. When getting a record, the `_version` field will tell how many times the record was changed.

All documents have automatically set IDs in Elasticsearch, that can be used to find for a specific object in the database. This is not most common use-case, though.

Data is accessible directly through the REST API that Elasticsearch makes available when running. Access through the Java API is also available, as well as for Javascript, Ruby, and other common web-oriented languages.

The main way to use Elasticsearch is via the mentioned REST API. The options that the API brings are actually much more extense than what is going to be explained now, but this is enough to get some insight. For these examples, a sample dataset, provided in the "Getting Started" guide[27] in Elasticsearch's site, will be used. Note that the dataset is automatically generated, so results may differ.

Notice that all examples use `curl` Unix command. When not set, the request method (`-X`) is GET. All URLs are relative to the actual Elasticsearch server URL.

First, the wanted index has to be created. Elasticsearch creates them automatically by default when making the first request to that index. With the chosen example, all data is bulk-written at once from a JSON file. So, to see how a request to create data works, see the example in figure 5.

```
# curl -X POST /bank/account?pretty

{
    "account_number": 1254365,
    "balance": 1500,
    "firstname": "Unai",
    "lastname": "Perez"
}
```

Figure 5: Creating a record in index bank, type account, Elasticsearch

The response to that request, in JSON format as well, will tell if the record was successfully created (`created` field), which ID it has been given, and what version of the record it is, among other information.

There are two ways to search records. One is to write the parameters of the query as URL parameters, where they go after a question mark (?) and separated by ampersands (&). The cleaner method is to write the query in the request body, in JSON format. Figures 6 and 7 are examples of both methods, that get the same response.

```
# curl /bank/account/_search?q=*&sort=account_number:asc&pretty
```

Figure 6: Search with query string parameters (in URL), Elasticsearch

In the response body, it's possible to see how long it took to search and how many hits there were, among other information and the matching objects.

[27] Getting Started with Elasticsearch: `http://bit.ly/2i8syt3`

```
# curl /bank/account/_search

{
    "query": { "match_all": {} },
    "sort": [
        { "account_number": "asc" }
    ]
}
```

Figure 7: Search with request body parameters, Elasticsearch

Ultimately, Elasticsearch provides very strong search features, like suggesters, that suggest similar looking terms based on the used search terms, and tons of aggregation types and methods. These are too complex and require a more in-depth analysis, but are worth mentioning. Overall, Elasticsearch is a powerful software that provides great search capabilities, for when it's more important to retrieve (and how to retrieve it) than to just store.

Installing Download the appropriate version from its website[28] and extract it where wanted. If chose to download a `.deb` or `.rpm` package, open it with the SO's package manager.

Running If installed manually, just run `elasticsearch.bat` on Windows or `elasticsearch` on Linux, inside the extracted `bin` folder. If installed through package manager, manage the service with `service elasticsearch`, then using `start` and `stop`.

Using It's possible to use it through one of the many APIs, with Java, Javascript, .NET and more. Alternatively, it's possible to use it through its REST API, via Postman (for tests mostly) or a client-side app.

3.5.2 Apache Solr

Pronounced "solar", it's an open source, enterprise search platform. It's based on Lucene, like Elasticsearch. It was created by Yonik Seeley, at CNET Networks, as an internal project to add search capability to the company website. Then, in 2006, it was donated to Apache Foundation and made open source. Today, it's the second most popular search engine, just behind Elasticsearch.

Enterprise search refers to the source from where the engine searches (Refer back to the classification of search engines in section 3.5). Specifically, it searches in the internal network of the company. To give some actual example, Netflix uses Solr to search the movies stored in their network, and Instagram to search for hashtags, pictures and users.

[28] Download Elasticsearch: `http://bit.ly/2i8GnYw`

Still, Solr runs as a full-text search engine. It indexes text from all kind of files: From Office files like `.doc` or `.ppt` to JSON and XML. Among other features, it's worth mentioning that Solr is designed for scalability and fault tolerance, and also offers REST XML and JSON APIs. Also, Solr's REST API converts the data to the desired format, these including common formats like JSON and XML, but also PHP or Python arrays.

Data is classified in collections, which work in a similar way to tables in relational systems, but with no relationship between them. When first running Solr, it creates an empty collection called `gettingstarted`. That's the collection used in the later examples.

A web application is provided when the server is up. This application can be used to manage the cluster and the shards individually, to make sanity checks, or its interface for querying can be used to make some tests. Interacting with data is done via the REST API, but the web app's interface helps learn the parameters and their functions.

For the examples that follow, the `curl` Unix command will be used. If no request type (`-X`) is specified, it is assumed to be GET, and all paths are relative to the collection's actual address, which is: `http://<serverIP>:<port>/solr/gettingstarted`.

To post documents to Solr, a script, called `post`, is provided in the `bin/` folder of Solr's installation directory. Say the goal is to insert all documents in path `/usr/docs`. Then, the command on figure 8 has to be issued.

```
$ bin/post -c gettingstarted /usr/docs/
```

Figure 8: Posting documents, Solr

Once Solr is populated, searching is the next step. To search for all documents that are stored, check figure 9. By default, Solr returns only the first 10 hits of the search. That can be changed with query string parameters `start` to define from which index to start returning, and `rows` to set how many hits are wanted.

```
# curl /select?q=*:*&wt=json
```

Figure 9: Getting all documents, Solr

The `wt` parameter sets the format of the response, which accepts, as mentioned before, XML and Python among others. The `q` parameter belongs to the actual query, the words that are being searched for. If the objective is to find all documents the title of which includes the word "Class", but only want to receive the title and content type, check figure 10. It's important to mention though, that Solr is case-sensitive by default.

```
# curl /select?q=title:*Class*&fl=title,content_type&wt=json
```

Figure 10: Getting documents under conditions, Solr

It's also possible to search for documents that have certain words that don't have to be one after the other. Or to search for documents that include one word, but do not have another word. For example, taking example in figure 10, say the goal is to exclude all classes from the API that are deprecated. Check figure 11 for that. It's also important to say that the URL must be correctly encoded.

```
# curl /select?q=+*Class*%20-*Deprecated*&wt=json
```

Figure 11: Excluding words from the search, Solr

It must be said that if no asterisks (*) are added to the beginning and the end of the search term, Solr would search for exact coincidences, rather than searching a title that includes the search term. They work the same way as percentage symbols (%) with `LIKE` clauses in SQL.

By only reviewing the most simple features of Solr, it's clear that its searches can be a very powerful form of information retrieval. It's also very easy to learn and start using.

Installing Download it from the Apache's mirror site[29]. Extract it where wanted.

Running In the installation directory's `bin` folder, run the command `solar start -e cloud -noprompt`. Solr will start two nodes that listen to two different ports, and will create test collection `gettingstarted`.

Using Post data with the `bin/post` script, as explained earlier. Then, data can be interacted with via the REST API or the web application, in the address and port that the starting script specifies.

[29]Solr mirror site: `http://bit.ly/2jdYqwh`

4 Scalability in NoSQL

Scalability, in this context, refers to the ability to run several synchronized instances of the database server, in one or more machines. The goal to this is to bring more power to the database and the application using it, and therefore faster response time and power and data balancing.

To be more specific, vertical scalability is the ability to run several interconnected instances within the same machine, while horizontal scalability refers to running interconnected instances in different machines. Each instance that runs in each machine is called a node.

Two general strategies exist to achieve horizontal scaling. One is replication, which means that a copy of the whole database will be stored in each of the machines. The load is then balanced between nodes. The other strategy is sharding or partitioning, which consists on dividing the data. Depending on the database type, this strategy is approached in different ways.

In general, NoSQL is said to be the replacement of relational databases in terms of scalability, like Pokorny in [11]. While it's actually harder to scale relational databases (in favor of other features), it's not at all impossible as it is sometimes said. That is what is going to be discussed now.

When it comes to relational databases, sharding would mean to either relocate some tables in other nodes, or to divide the tables to store some of the data in them on different nodes. In both cases, joins would get much more costly, because in some cases they may involve communicating between nodes. Transactions that block tables that are in other nodes would cause trouble as well.

That's why in some cases it may be better to choose replication instead. It's done by following the master-slave architecture. Writes are done to the master, which replicates them to all slaves. Then, reads are done to the slaves.

Redis, as other **key-value stores**, supports partitioning of the data. Partitioning in this case, as no relationship exists, can be done easily. It just consists on storing the key-value pairs in different nodes. Redis particularly has two strategies to know what pairs go to which node (so that it is then able to locate each pair), one regarding to id number ranges and the other regarding hashing the key string into a number.

Document-oriented databases work well with sharding too. Documents are independent pieces of data. So, storing it in one node or in the other makes no difference. Specifically, MongoDB provides a software, called Ops Manager, that configures all about sharding automatically. This technique used

by MongoDB is called auto-sharding, a technique that Liu et al research in [12]. Partitioning would also be an acceptable way to scalate.

Sharding gets more complex with **graph databases**. These also have relationships that may point to nodes (as in graph database entity) that are stored in other machines. It would be very convenient to store independent chunks of data that don't have any relationships, this is, subgraphs, in different machines to avoid that problem. Replication, though, would not cause any problem.

Finally, **column-oriented databases** aim at storing lots of information while being efficiently horizontally scalable. For example, Apache Cassandra, which is currently the most popular of this kind of databases, allows a seamless addition of more operating nodes, that increases performance linearly.

All in all, the system that makes it easier to scale horizontally will win the game, as that is what makes the real difference when looking for good performance with big amounts of data. All the presented types of databases implement in one way or another the possibility to scale both horizontally and vertically. Whether it's straightforward to actually set the nodes up and make the structure to work, along with how the system's approach to storing data matches the purpose of the project, will determine if the system is chosen or not.

5 What and when to use

If something, so far it could be deduced that NoSQL databases, rather than being just better than relational ones, are just most suitable in some scenarios. Even some types of NoSQL databases suit some cases better than others. What follows is a little review with examples of which database type to use in what case.

One possible use of a **key-value store** is as the back-end database of a web application. Usually, when users enter a website, it gives them a unique cookie value with which is going to identify all the session related information of each user in the database. Even if the user enters just once and then leaves, this cookies value is going to stay in the database. It will take space to keep the information in there and this is critical in a RDBS.

If the web retrieves all the information every time a new page is loaded, this means a new query every little time. One entry might make no difference, but when a lot of information is stored, the performance will drop. A KVDBS fits this job better, as a regular query is much faster, and the implementation much simpler. Moreover, some of these systems have an expire command that will remove the key automatically once the expiration date has been reached.

In one case in which information could fit a relational schema, but relationships are more flexible and optional, and therefore information would actually bear differences, **graph databases** are the best choice. Also, as such databases offer recommendation queries (see appendix A), it may be suitable for even more use cases.

For the sake of quoting a real case, there's the case of Neo4j's client Shutl. Before being bought by eBay, Shutl was an online marketplace that offered same-day delivery of all kind of purchases. To do such thing, it searched for the courier that was closer to the delivery point using MySQL joins. When data started to grow, this solution dropped performance and began to be too difficult to maintain. The domain was modelable in the graph database, solution which in turn offered more speed and ease of use. So, MySQL was replaced by Neo4j.

Another example would be a database about football, in which players, teams, leagues, fixtures and games will be stored. All entities and relationships are easily distinguishable, and it's hard to find a relationship that can't be predicted. In other hands, there is no need of flexibility. In such case, a relational database may be a even better suit than a graph database.

Document-oriented databases differ a lot from relational databases in the way data is stored and interpreted. DODSs are much more flexible and straightforward. Also, some underlying features that relational databases offer, are not offered by document-oriented databases. For example, high insert rate is preferred over transaction safety. Also, scaling is much faster and easier, and bigger amounts of data can be handled. In contrast, a single MySQL table's performance will drop when crossing 5GB of size.

Say the goal is to store a catalog of all products in a shop. All products are very different, and therefore, different information has to be stored. Imagine this in a relational table. It would require either a table per each, which is unfeasible, or a "flexible" table design, in which many columns would be null, depending on the product type. Null values still take up space and slow down overall speed. So, for this case in which flexibility is needed, a document store would be more suitable.

Now, to shortly address **column-oriented databases**, it's mentionable that the main difference with a regular relational database is how it stores the information in disk. It is not at all that different from a document-oriented database as well. Generally, when a document is used, when retrieving it back all the document is gotten. On the other hand, with column-oriented databases, it is more common to refer to specific sets of columns. Also, reading is much faster in a column-oriented database, than it is in a document-oriented one.

Finally, **search-engines**' actual purpose is closer to search and find than it is to plainly store information. For example, consider this example took from Elasticsearch's website. The goal is to run a price alerting platform that allows customers to specify a rule saying they are interested in a specific product, and that they want to be notified when the price falls below a given price. Then, prices have to be pushed to the engine, use a reverse-search to match price changes with customer rules, and then alert the matching customers.

The NoSQL types are ambiguous. Each has its own description, but between the description and some actual implementations of the databases, there may be some big differences. Multi-model databases too exist, which offer more than one type of NoSQL implementation within the same system. The first step before choosing one system over others is to consider what information type will be stored, if it needs flexibility or not, and if speed matters more than security.

6 Live testing

A very common usage of databases in general, and some of the mentioned database systems in particular, is web applications. Back-end applications, a growing concept with growing technologies like Node.js or Spring Boot, are used to retrieve the data from the databases.

Back-end applications are the ones that perform the CRUD operations (***C**reate **R**ead **U**pdate **D**elete*) on the database. They parse the information into JSON documents usually, though some use XML as well. This way, the front-end applications handle interpreted information instead of having to parse it themselves. The workload is split and everybody wins.

The most common back-end approach is called REST API. It has become a standard in the late years, though the concept has been wandering around since 2000, when it was first made public by Roy Thomas Fielding in his thesis [13].

To create a realist environment to test the databases, an API has been built using Spring Boot (Java) for each database. When possible, two APIs have been built for one database, one using the libraries provided by the developers and other one using the Spring framework dedicated to the database.

Gatling testing framework will then be used to test the database APIs. Gatling works by sending HTTP requests to the server according to the simulation code (written in Scala). It's possible to code 10 virtual users to try to access the database at one given time, or it's possible to code hundreds of virtual users to eventually access through a given time lapse. Then, Gatling will

generate graphics comparing the request amount and the response time, thus giving a good retrospective of how the databases work.

The source code can be found on Github[30], along with a readme file with all the steps needed to make it work.

7 Conclusions

The DB-Engines Ranking [31] shows that, nowadays, the most popular databases are of the relational type. As proven in this paper, the so-called NoSQL databases are indeed very different from the SQL most developers are used to learn and use. That may be one of the most important reasons as to why they still lead the market.

Still, NoSQL definitely are a powerful alternative to relational databases. Some types may fit some use-cases better than relational systems. So it's not only about a different approach on how to access data, it's also about a different approach on how to interpret data.

After analyzing these NoSQL systems, they can be divided in two groups depending on how they see data. On the one hand, **graph databases** are the only type of NoSQL systems that belong to the **structured** data group, to which also belong relational databases.

On the other hand, the other types of NoSQL databases belong to the **non-structured** data group. These means that they store either **unstructured** or **semi-structured** data. One example of unstructured data could be the case of key-value stores, which store single values with no correlation with each other whatsoever. Semi-structured data is the case of, for example, document-oriented databases, that store objects of internally structured data like JSON or XML.

In table 3, some attributes of the different NoSQL types are compared, so that the most common questions can have a quick and brief answer.

When it's time to choose one over all the available options, it's crucial to stop and think how the data can be interpreted, keeping in mind the classification just mentioned. But it's also important to think what of all features is more valuable for the use-case. Is availability more important than real time consistency? Is more important to read fast than it is to write fast? Is it necessary to scale horizontally?

[30] NoSQLTester on Github: `https://github.com/unaipme/NoSQLTester`
[31] DB-Engines Ranking: `http://bit.ly/JaDlgu`

	KVDBS	Graphs	DODBS	CODBS
Data structure	Non-structured data	Structured data	Semi-struct. data	Semi-struct. data
Horizontal scaling	Easy	Hard	Easy	Easy
Adaptation from SQL	Easy	Medium	Medium	Medium

Table 3: Quick comparison of NoSQL types

For all that questions, answers come with specific implementations of the systems, rather than with any system of a given type. The most popular databases of each type were mentioned in this paper, excluding the multi-model ones. So it's important to know that maybe other implementations took more effort into bringing a feature that you are looking for.

It's easy to see why these systems had the rise in popularity that they had. They serve for countless use-cases that relational databases did not fit. They are mostly still young technologies that are in development, but it's definitely worth keeping track of the already popular ones, and keep an eye open for the new technologies that will emerge on the years coming.

References

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[2] Julian Browne. Brewer's cap theorem. *J. Browne blog*, 2009.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[4] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[5] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[6] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, February 2008.

[7] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proc. VLDB Endow.*, 2(2):1664–1665, August 2009.

[8] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[9] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[10] Emanuel Goldberg. Statistical machine, December 29 1931. US Patent 1,838,389.

[11] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[12] Yimeng Liu, Yizhi Wang, and Yi Jin. Research on the improvement of mongodb auto-sharding in cloud environment. In *Computer Science & Education (ICCSE), 2012 7th International Conference on*, pages 851–854. IEEE, 2012.

[13] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

Appendices

A Getting recommendations from Neo4j

Starting from the example movie database that is used for the testing, the goal is to find out who can introduce Tom Cruise to Tom Hanks. For that, the query written in Cypher on listing 14 can be used.

```
MATCH (hanks:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)
<-[:ACTED_IN]-(coActors),
(coActors)-[:ACTED_IN]->(m2)
<-[:ACTED_IN]-(cruise:Person {name: "Tom Cruise"})
RETURN hanks, m, coActors, m2, cruise;
```

Listing 14: Finding common co-actors for T. Hanks and T. Cruise, Neo4j

The result reveals that both Hanks and Cruise have worked with Bonnie Hunt, Kevin Bacon, and Meg Ryan, so it must be one of those who introduces them both.

To have a better insight of what the query does: It finds the node with tag `Person` that has Tom Hanks as name, then finds all nodes with which the previous node has a relationship with tag `ACTED_IN`. Then, it just follows down the relationship chain, and finds all people that have acted with both Tom Hanks and Tom Cruise at any time.

To put a more simple example, in a similar way, imagine digital videogame retailer Steam's database, that gives suggestions on what game to buy next based on what games the user likes. Consider the dataset in listing 15.

Now, knowing what games user with name "Unai Perez" likes, the query in listing 16 returns a list of games he may also like, based on what games like the people that also likes those games, which in this case it will be The Binding Of Isaac: Rebirth and Doom. It also returns a string that says because of what game is the first game being recommended.

B CRUD operations in DynamoDB

As seen in DynamoDB's section, the table creation is made by creating a Javascript object with certain attributes that define the table, then passing the object as an argument to the `createTable` method. For the examples that follow, the example table created with the script in listing 17 is going to be used.

```
CREATE (up:User {name: "Unai Perez"}),
(mr:User {name: "Mikel Retolaza"}),
(pl:User {name: "Pablo Lombardo"}),
(au:User {name: "Aitor Urrutia"}),
(doom:Game {title: "Doom"}),
(limbo:Game {title: "Limbo"}),
(gungeon:Game {title: "Enter the Gungeon"}),
(isaac:Game {title: "The binding of Isaac: Rebirth"}),
(fc3:Game {title: "Far Cry 3"}),
(up)-[:LIKES]->(fc3), (up)-[:LIKES]->(gungeon),
(mr)-[:LIKES]->(fc3), (mr)-[:LIKES]->(doom),
(pl)-[:LIKES]->(gungeon), (pl)-[:LIKES]->(isaac),
(au)-[:LIKES]->(limbo)
```

Listing 15: Example dataset for Steam, Neo4j

```
MATCH (g:Game)<-[:LIKES]-(u:User)-[:LIKES]->(g2:Game)
<-[:LIKES]-(op:User {name: "Unai Perez"})
RETURN g, "Because you like " + g2.title;
```

Listing 16: Getting game suggestions, Neo4j

Inserting data in the database works in a similar way. First, an object with all the parameters and the data to be inserted must be defined, and then that information must be passed to the method `docClient.put`. A full example is in listing 18.

The object that is needed to put a document in the database needs to attributes: The table name, and the item with all the key-value pairs that have to be stored. In the item object, the primary keys defined for the table also must have a value. That is the case of the previously mentioned example, in which the attribute `Id` has a value.

To retrieve back the object that was just inserted, the process is similar to inserting. To get objects from the database, define a parameter object in which the table from which to get and the conditions to search are set, like in listing 19. In that example, all documents from table `People` that have the value `'Unai'` for `FirstName` will be retrieved by method `get`.

There exist more methods, like a method to batch-write many items, a method to scan everything in a table, and more. All this can be learned in DynamoDB's web application. Issuing the function `tutorial.start()` starts a tutorial that explains everything explained in this brief appendix and more.

```
var params = {
    TableName: 'People',
    KeySchema: [ {
            AttributeName: 'Id',
            KeyType: 'HASH'
    } ],
    AttributeDefinitions: [ {
            AttributeName: 'Id',
            AttributeType: 'N'
    } ],
    ProvisionedThroughput:  {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
};

dynamodb.createTable(params, function(err, data) {
    if (err) {
        // handle error
    } else {
        // do something
    }
});
```

Listing 17: Example table, DynamoDB

```
var params = {
    TableName: 'People',
    Item: {
        Id: 1,
        FirstName: 'Unai',
        LastName: 'Perez'
    }
};

docClient.put(params, function(err, data) {
    if (err) {
        // handle error
    } else {
        // do something
    }
});
```

Listing 18: Putting data, DynamoDB

```
var params = {
    TableName: 'People',
    Item: {
        // add here the conditions to get
        FirstName: 'Unai'
    }
};

docClient.get(params, function(err, data) {
    if (err) {
        // handle error
    } else {
        // do something
    }
});
```

Listing 19: Getting data, DynamoDB

C Setting HBase up

It is not required to install Hadoop for HBase to work, so the instructions here will manage without Hadoop.

First, download the latest HBase version from their website, as explained in its own section. Extract it where more appropriate.

Second step is to check that `/etc/hosts` text file contains a line that refers to IP 127.0.0.1 as `localhost`. If there is no reference to `localhost`, add the following line at the beginning of the file: `127.0.0.1 localhost`.

Next step is to go edit the file `conf/hbase-site.xml` inside the extraction folder of HBase. Before doing it, decide where the data has to be stored in the machine. One directory for HBase data and another one for Zookeeper are required. When decided, add the lines on figure 12 inside the `configuration` tags of said file.

Then, follow the instructions to run HBase, explained in section 3.4.1.

D Using HBase shell

Once HBase is correctly configured, it's possible to start using the shell to manage the data in the database. The first step after a clean install, is to create a table. To do that, the `create` command must be used, as shown in listing 20.

```
<property>
    <name>hbase.rootdir</name>
    <value>file:///DIRECTORY/hbase</value>
</property>
<property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/DIRECTORY/zookeeper</value>
</property>
```

Figure 12: Simple configuration, HBase

```
create 'tablename', 'columnfamily1', 'columnfamily2'
// with versions:
create 'tablename', {NAME=>'columnfamily1', VERSIONS=>5}
```

Listing 20: Creating a table, HBase shell

Tables can be enabled and disabled, this is, the use of the tables can be individually allowed or disallowed without deleting the table. This is done with commands `enable` and `disable`. Tables must be disabled to be able to delete them, and are automatically enabled when created. Check listing 21.

```
enable 'tablename'
disable 'tablename'
drop 'tablename'
```

Listing 21: Enable, disable and delete tables, HBase shell

Columns don't have to be created, only column families must be defined when creating the table. To put data into a table the command `put`. To retrieve some specific data from the table, command `get` is used. To get all data of all versions from one table, `scan` is used. Refer to listing 22 to see examples of these commands.

E HBase Java API examples

Developing using HBase's Java API can be tedious, as most of the tutorials and examples on the Internet are already deprecated and, also, the libraries can cause some incompatibility with other libraries.

That is the case of Google's Guava. To put it shortly, HBase has a dependency to a Guava library that is compatible. But if another dependencies are added to the project's POM (or equivalent with dependency management

```
create 'people', 'fullname', {NAME=>'address', VERSIONS=>3}
put 'people', 'uperez', 'address:city', 'Ordizia'
put 'people', 'uperez', 'fullname:firstname', 'Unai'
put 'people', 'uperez', 'fullname:lastname', 'Perez'
// ... //
get 'people', 'uperez', {COLUMN=>'address:city'}
get 'people', 'uperez', {COLUMN=>['address:city',
    'fullname:firstname']}
get 'people', 'uperez', {COLUMN=>'address', VERSIONS=>3}
scan 'people'
```

Listing 22: Putting, getting and scanning a table, HBase shell

software other than Maven), a dependency to other higher Guava versions may be included. The higher versions always prevail, if not explicitly set in the POM. The general recommendation is to add Guava 15.0 to the POM explicitly, to avoid any problem.

Another inconvenience with the API is that it works with byte arrays rather than with regular strings. As it can be seen in the example sources, most strings have to be converted to byte arrays using the API's `Bytes` utility class. This makes the code look much more messy.

Also, all tests have been done running both the Java code and the HBase server in the same machine. These examples won't show how to connect remotely through Java API.

To create a connection to local HBase server, refer to listing 23. It shows two approaches. If the connection is needed for a single isolated function, the first option is the best suiting one. If the connection is needed for an undefined time, or if the connection takes too long to make, then the second approach is the most appropriate.

HBase Java API offers two types of operations. Operating directly with the connection offers CRUD operations with the database. But the connection object provides an `Admin` object with which administration operations like creating or destroying tables can be done.

To create a table in HBase, look at listing 24, in which it is shown how to create a table with two column families. To disable and delete a table, take a look at listing 25. Finally, to write and retrieve the data back from the database, refer to listing 26.

```java
// This way, the connection will automatically close
// But it's also less flexible
Configuration config = HBaseConfiguration.create();
try (Connection conn=ConnectionFactory.createConnection(config)) {
    // do something...
}

// This way it can be accessed at any point in the class
class HBaseOps implements AutoCloseable {

    private Configuration config;
    private Connection conn;

    public HBaseOps() {
        config = HBaseConfiguration.create();
        conn = ConnectionFactory.createConnection(config);
    }

    @Override
    public void close() {
        conn.close();
    }

}
```

Listing 23: Creating a connection with Java API, HBase

```java
Admin admin = conn.getAdmin();
HTableDescriptor htable = new HTableDescriptor(TableName.valueOf("people"));
htable.addFamily(new HColumnDescriptor("address"));
htable.addFamily(new HColumnDescriptor("fullname"));
admin.createTable(htable);
```

Listing 24: Creating a table with Java API, HBase

```java
TableName tablename = TableName.valueOf(Bytes.toBytes("people"));
Admin admin = conn.getAdmin();
admin.disableTable(tablename);
if (admin.isTableDisabled(tablename))
    admin.deleteTable(tablename);
```

Listing 25: Disabling and deleting a table with Java API, HBase

```
Put put = new Put(Bytes.toBytes("uperez"));
put.addColumn(Bytes.toBytes("address"), Bytes.toBytes("city"),
    Bytes.toBytes("Ordizia"));
Table table = conn.getTable(TableName.valueOf("people"));
table.put(put);
Get get = new Get(Bytes.toBytes("uperez"));
get.addColumn(Bytes.toBytes("address"), Bytes.toBytes("city"));
// Optional: Set how many versions are wanted
get.setMaxVersions(3);
Result result = table.get(get);
byte [] citybytes = result.getValue(Bytes.toBytes("address"),
    Bytes.toBytes("city"));
String city = Bytes.toString(citybytes);
```

Listing 26: Putting and getting data with Java API, HBase

F CRUD operations in MongoDB

MongoDB's client allows to manage all data directly using Javascript code with BSON, its propietary JSON-based object notation. MongoDB stores data in collections, which belong to databases. When connecting via the client, the first thing to do is to set the database in which to operate. Issue the command `use <dbname>` for that. The command `show collections` will show all the collections created in the database.

Once in the wanted database, and knowing which collection is going to be used, first thing to do is to insert some data. For the examples, database `test` and collection `restaurants` are going to be used.

Inserting one restaurant is as easy as passing a Javascript object, with the wanted key-value pairs, to the method `insert` of the collection. When inserting, an ID field can be manually defined using constructor `ObjectId`. Even if defined to another field, Mongo will automatically create an `_id` field with an `ObjectId`. See listing 27.

After inserting some documents, all data can be retrieved back using collection's `find` method without parameters. If the `pretty` method of the result JSON is called, the JSON will be shown properly indented. This can be seen in listing 27. To get only documents that match certain search criteria, like the `WHERE` clause in SQL, a Javascript object with all conditions has to be passed to the `find` method, as seen in listing 28.

In listing 28 in particular, the query asks for all documents in collection `restaurants` that are from the specified borough, and that were opened later

```
db.restaurants.insert({
    borough: "Garagartza",
    cuisine: "Gastroteque",
    name: "1990",
    opened: 2015,
    _id: ObjectId()
});
db.restaurants.find().pretty();
```

Listing 27: Inserting a restaurant, then getting all, MongoDB

```
db.restaurants.find({
        borough: "Garagartza",
        opened: { $gt: 2010 }
    }
).pretty();
```

Listing 28: Conditioned select, MongoDB

than 2010. Operators, which are the keys starting with dollar sign ($), make available to use the same functionality as with <and >operators in SQL.

Defining several fields in the find parameter object means they will be treated as `AND`. Though, both `AND` and `OR` have a explicit way to be written. Refer to listing 29, in which all restaurants that were opened in 2010 or 1994 and whose cuisine is not "Pintxos" are found.

```
db.restaurants.find({
    $and: [
        { $or: [
                {opened: 2010},
                {opened: 1994}
        ] },
        { cuisine: { $ne: "Pintxos" } }
    ]
}).pretty();
```

Listing 29: Using OR and AND, MongoDB

These condition parameter objects used in the previously mentioned examples also serve to update or to delete documents. To update, all documents that match the condition object are updated matching the updated data object, passed as second parameter. To delete, all documents matching the condition

object are deleted. See listing 30.

```
db.restaurants.update({
    $or: [
        {opened: 2010}, {opened: 1996}
    ]
}, {
    $inc: {opened: -1}
});
db.restaurants.remove({name: "1990"}, 1);
```

Listing 30: Updating and deleting documents, MongoDB

In the mentioned listing, in the deleting example, the second parameter is `1`, meaning that only the first matching document is to be deleted.

When updating, there is much more to it than what it seems. If a regular object with key-value pairs is used as updating object, the objects that match the conditions will be completely replaced by the new object. To update just certain values, the update operators[32] are used. The example on listing 30 uses `$inc`, the function of which is to add the specified number (positive or negative) to the selected value.

MongoDB has much more functionalities than the few mentioned in this appendix. It provides aggregating operators, which are the equivalent of SQL aggregation functions, such as `COUNT` or `SUM`. It also provides sorting, limiting and projection (which translates to choosing columns to select in SQL).

[32] MongoDB update operators: `http://bit.ly/2i9P5Ww`