UNIVERSITÉ LIBRE DE BRUXELLES

Faculty of Science

INFO-H415 - Advanced database

Document oriented Databases

Alain Issa, François Schiltz

12 octobre 2015

Academic year 2015-2016

Contents

Chapter 1

Introduction

During the rise of the World Wide Web, new problems for the storage of data appeared. Indeed a few years ago a thousand of users for an application was a huge number and we were not expected to have more than a few thousand . But now billions and billions of people are connected through the Internet and the number of data to store has increased significantly,this problem is related to Big Data problem.

To manage more easily this huge amount of data, it becomes a necessity to structure the data stored, that is why databases has been created [1].
This picture (figure 1.1) shows the problem we would meet today if we manage data without structured model [1].

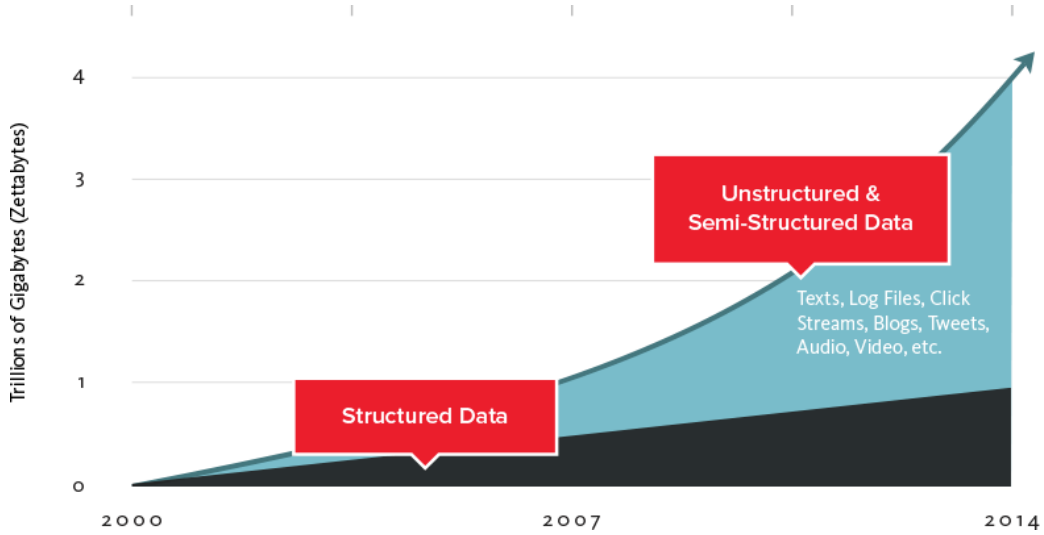

Figure 1.1: Comparison between the size taken by structured and unstructured data with the Big Data problem. Unstructured data takes more space if they are stored

Nowadays new paradigm of database were developed to improve the web access of data like JSON and XML pages. These type of database are called documents oriented databases.

Through this report we are going to give some informations about document oriented databases and especially what is it and what are the main differences between this type of database and a more traditional relational databases.
Finally we will present an implementation of document databases and how it should be implemented.

[1]Picture found on the following website : `http://www.couchbase.com/nosql-resources/what-is-no-sql`

Chapter 2

State of the art

2.1 What is a document database

A document database is one type of modeling data in the NoSQL(Not only SQL) family databases that we are going to present briefly in the next subsection.
This model is based on the paradigm key-value. In our case the value is a document of type JSON or XML. So with one key we can get a structured set of information (see the Figure 2.1 and bibliography number [10] [1]) and to do the same operation in the relational model we need to do many join that reduce the performance.
In other word a document database takes the data you want to store and aggregates it into documents using, for instance, the JSON format. So in a single document we can have the same information that is usually stored in many table in a relational model as we can see in the figure 2.2 [2].

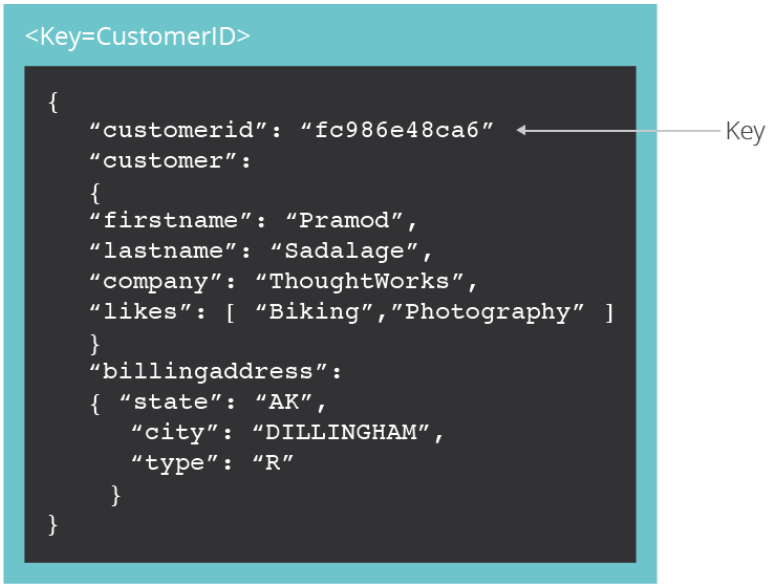

Figure 2.1: A document oriented database with the key and its value

[1] Example of document database taken on the following website: `https://www.thoughtworks.com/insights/blog/nosql-databases-overview`
[2] Picture taken on the following website: `http://www.couchbase.com/nosql-resources/what-is-no-sql`

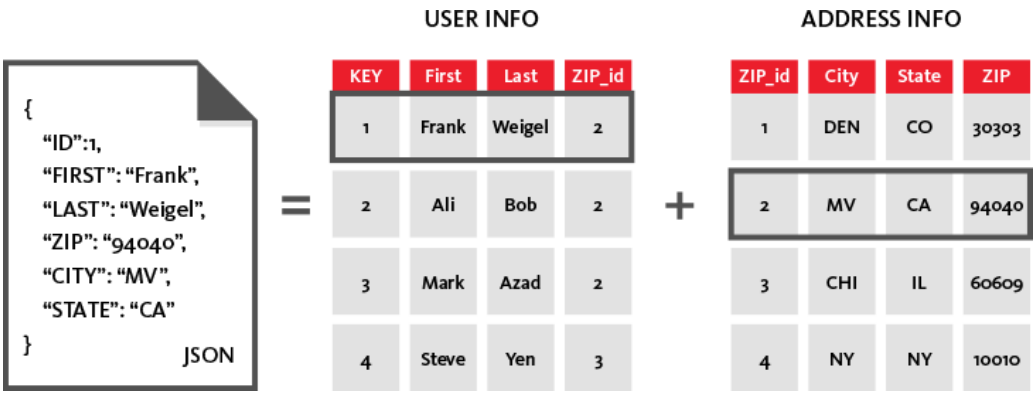

Figure 2.2: Document-oriented Database: How can be data from relational model stored in document-oriented model (here it's in a JSON format). This approach provides better data model flexibility, greater efficiently in distributing documents, and superior read and write performance.

2.1.1 NoSQL databases

This acronyme was coined in 1998 and the main idea behind NoSQL is to try to make databases model without using SQL and tabular relations like it's the case in the relational model [2].These two types of models are very different not only in their structure but also in their way to access the data.

Indeed, the relational model takes data and separates it into many interrelated tables that contain rows and columns. Tables reference each other through foreign keys that are stored in columns. And with this structure when we looking up for data, the desired information has to be collected from many tables and combined before it can be provided to the application. On the other hand when we have to write data, the writing action has to be performed on many tables.

There are a lot of different NoSQL databases [3] (other than document oriented) and each one of them has a different structure for example [5]:

- Graph stores that are designed for data whose relations can be represented as a graph (elements interconnected with an undetermined number of relations between them). An example of the kind of data that can be represented by this kind of model could be social relations (as in Figure 2.3 [3]), public transport links, road maps or network topologies.

[3]Graph example taken on `https://www.thoughtworks.com/insights/blog/nosql-databases-overview`

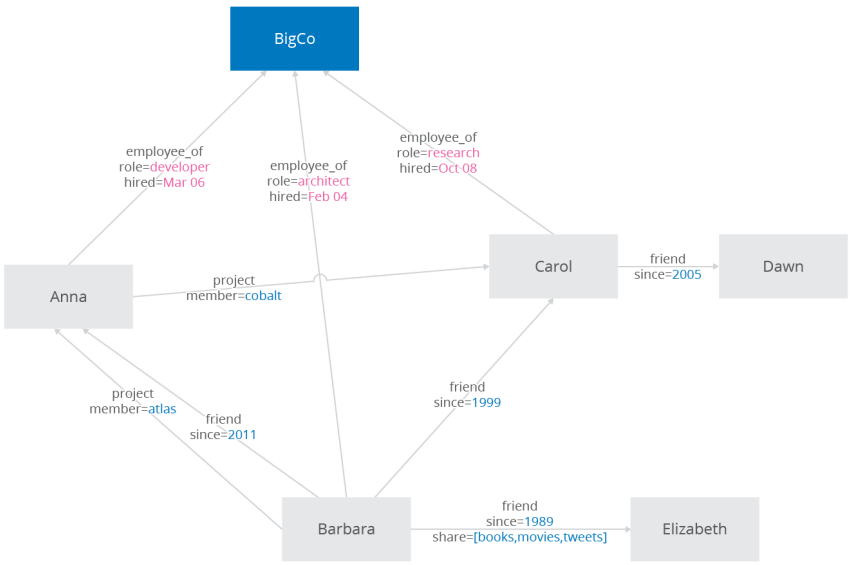

Figure 2.3: Social relations represented by a graph database

- Key-value stores are very simple because it is basically based on a dictionary (map) where the data are represented by a pair key/value.
- Wide-column stores is similar to relation table except that in a NoSQL database the number of column are dynamic in contrast to relational table where column are fixed on the creation of the schema. So in a wide-column stores database when we want to add a data the number of column may vary , it avoid to return column with NULL value.
- Document store can look similar to wide column in the sense that they share the schema-less approach but the implementation is different. In a document database, each record is called a document and is stored separately.

2.1.2 Why choose NoSQL?

There are many argument we can provide to support the choice of choosing NoSQL [4].

- Performance
  For writing and reading to the database, a document store is faster than a traditional SQL database. For example, CouchDB can be 20 times faster than Postgres [6]
  This is due to the simpler nature of document, the database just needs to store directly the document and for a read just have to make a look-up for the data while for a relational query, it needs to decode the query and look in all the required table and aggregate the result. Thus, the simplicity of document is a real advantage for performance oriented data storage.
- More Flexible Data Model due to schema-less data model.
  A document database doesn't use a fixed schema, instead it can store any document it needs created at runtime. This allows the database to be modular and flexible, the application can store everything it needs and in case the business logic of a program change, the database doesn't need any additional change. Every document contains it own schema and a collection of document can have different schema. A single database can also work for multiple kind of application because all the data logic are on the client side.
- Better scalability than a relational model
  A lot of NoSQL database are designed with scalability and large amount of access in mind and

document store is no exception. Due to their simple nature, they allow to easily increase the database and split it between different server or node without having to increase the power of the server, this allows the database to run on simpler hardware for the same performance and to improve the reliability.

2.1.3 Performance overview of different databases

Before going further in this report let us define some notions [7].

Definition 1. "Database performance can be defined as the optimization of resource use to increase throughput and minimize contention, enabling the largest possible workload to be processed."

Definition 2. Scalability [8] is the ability of a database to grow to very large size while supporting an ever-increasing rate of transactions per second and to maintain its functionality and performance in case of high demand.

Definition 3. Flexibility is the ability for the solution to adapt to possible or future changes in its requirements with minimal modification of the system.

Definition 4. Complexity is degree to which a system or component has a design or implementation that is difficult to understand and verify.

Definition 5. Functionality is the sum of the different operations and abilities that a software can perform.

Now we have introduced some definition and presented some stores let us compare them [3]:

Data Model	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value	high	high	high	none	variable (none)
Column Oriented	high	high	moderate	low	minimal
Graph	variable	variable	high	high	graph theory [4]
Relational	variable	variable	low	moderate	relational algebra [5]
Document Oriented	high	variable (high)	high	low	variable (low)

Figure 2.4: Comparison of a few Data model based on five criteria

2.1.4 Why a document store

A document store is recommended when the application doesn't need the complexity of SQL and only store simple data without a lot of relation. It's also a good system when the application needs the speed without compromising reliability and scalability.

[4]The functionality of the graph database is limited to our knowledge of graph theory
[5]The functionality are defined by the relational algebra

2.2 How does it work

2.2.1 Data storage

- XML based
  In a document store, XML based data store uses XML files to store the data on a disk, a document.

  Example of XML document:[6]

```
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>
        Don't forget me this weekend!
    </body>
</note>
```

- JSON based
  In this case, the database use JSON files instead of XML. These kind of storage are better when the output needs to be usable for a web application or for programming object storage because JSON tools are included in all modern web browser and it can also be used to represent an object from every object oriented programming language.

  Exemple of JSON file:

```
{
  "docs":
  [
  {
    "_id": "562e35cf7f2d7ead13ac9188",
    "index": 0,
    "age": 40,
    "eyeColor": "blue",
    "name": "Reeves Pruitt",
    "gender": "male",
    "whishes": {
      "fruits": "Avocado",
      "number": 195
    }
  },
  {
    "_id": "562e35cf51a04e5226fa09e7",
    "index": 1,
    "age": 20,
    "eyeColor": "brown",
```

[6] From the w3c XML example: `http://www.w3schools.com/xml/note.xml`

```
        "name": "Dean Blankenship",
        "gender": "male",
        "whishes": {
          "fruits": "Melon",
          "number": 337
        }
      }
    }
```

- Other
  Some document store use other type of data storage, like MongoDB which works like a JSON based storage but transform its document to a binary format before writing to the disk to improve performance and reduce storage size [9]. CouchDB is a JSON database that use a B-tree for storage [11] [12].

2.2.2 Data querying and the map/reduce paradigm

For the retrieving of data, a document store is totally different than a relational database. It doesn't offer the power and complexity of those but still has a few tools that can be used by developer to retrieve specific set of data. One of those tools is the map/reduce paradigm. It's based around a two part operation and is optimized for taking advantage of parallel programming and multiple cluster database.

The map function will be run on each document and will return all the document or a subset of the document. This approach allows the query to be run simultaneously on multiple process and on multiple node of the database. All the result are grouped in a temporary data set.

After the map operation, the database will shuffle the data, it rearrange them by pregrouping them in a few data set and then will call the reduce method. The reduction of the data is an operation that will combine the data using a user specified rule and output the result. A reduction is done on each data set produced by the mapping so this operation is also done in parallel. After that an other reduction can also be done on the result if needed. [7].

The final result of the mapping and reduction will be stored in a special document called a view. A view can be used directly after a query like in a relational database or be kept in memory for a specified amount of time. This allow the database to precompute heavy query that don't need the absolute up to date resource and cache or keep a request that is often called. this allows to reduce the amount of work on the server by trading the consistency of the data.

[7] Map/reduce in CouchDB `http://docs.couchdb.org/en/1.6.1/intro/tour.html#running-a-query-using-mapreduce`

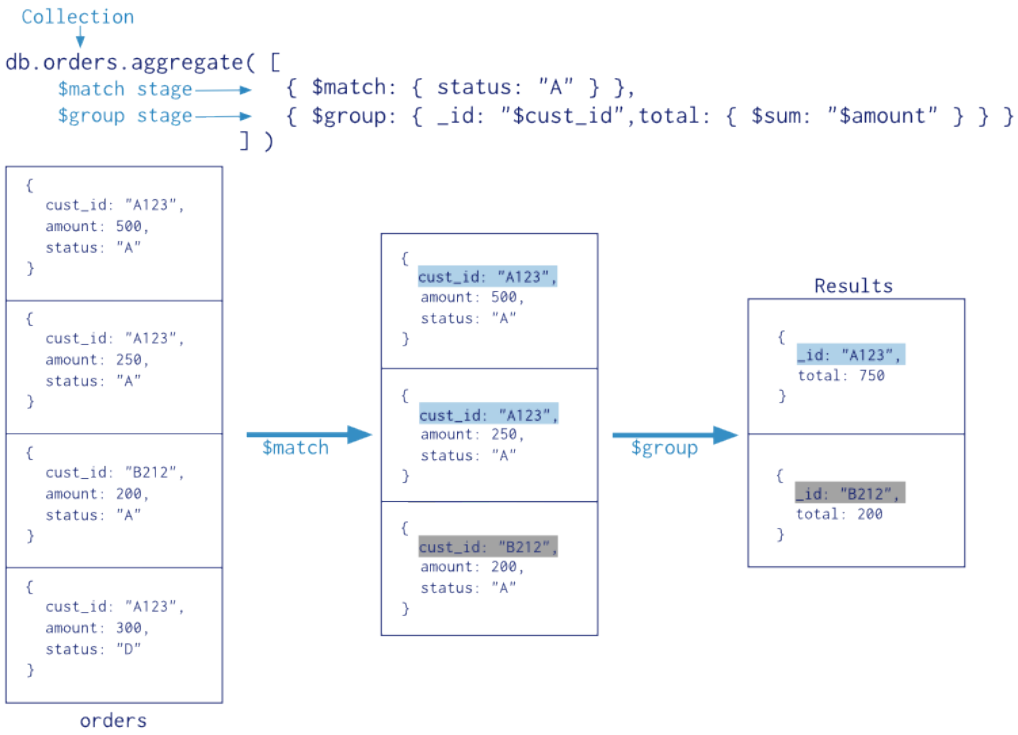

Figure 2.5: Visualisation of map/reduce operation from: http://docs.mongodb.org/manual/core/aggregation-introduction/

It needs to be specified that those two functions are loosely defined in the document database world so the implementation can be really different between two database implementation. For example, in CouchDB, the map function is responsible for deciding how to group the input and in RavenDB, it falls on the reduce function [13].

2.2.3 Inserting and Modifying

The insertion and modification details depends on the database and the file format choosen, so here we will explain specifically the implementation details of CouchDB query. On CouchDB, every deletion, insertion and modification are always an insertion on the database, no data is really deleted unless the user choose to remove a node of the storage Btree. This allow the database to maintain an history of all the modification done.

2.2.4 ACID

ACID (Atomicity, Consistency, Isolation, Durability), is a set of properties for a database.

- Atomicity
  In an atomic database, if a part of a transaction fails, the database will be left unchanged.
- Consistency
  Consistency ensure that every new state that the database takes is a valid states according to the design rules.
- Isolation
  This ensure that for parallel request on the database will left the data in the same states that it would have been if the requests were sequential.

- Durability
  The durability properties ensure that once a request has been made, it will remain even in the case of a power surge.

In the document store paradigm, the ACID properties are not granted and these some document store can be ACID compliant but not all. This is an important fact to consider when designing and developing a document store application.

2.3 The different solutions

2.3.1 Open source solution

- CouchDB
  Couch DB is a document store developed in 2005. It's now maintained by Apache since 2008. Couch use JSON file for storage and queries. It's mainly optimized for scalability and reliability. Couch is ACID compliant[8].

- MongoDB
  Mongo is the other principal open source database and is mainly optimized for speed and can be more than 25 times faster than Couch DB but doesn't offer the same reliability. This database is not ACID compliant because it only offer atomicity on each document and not on transaction level.

2.3.2 Proprietary solution

- Lotus Note
  Note (now IBM Note) is a client-server software that allow corporate management and collaboratives function between employees with mails, calendar, to-do list and others. The core component of the server is a document database, the first one tho be developed. Due to it's proprietary nature, not a lot is known about its implementation details.

- DocumentDB
  Document DB is a proprietary database developed by Microsoft that runs on their cloud platform. It's a JSON store.

- SimpleDB
  SimpleDB is the Amazon developed concurrent of Document DB. It's nearly equivalent but runs on the Amazon server.

[8]Couch DB properties: `https://wiki.apache.org/couchdb/Technical\%20Overview`

2.4 CouchDB

This section will cover CouchDB and it's advantage in an real world application and how to use it. It wont cover the technical part of the storage and everything that was already explained in the previous section.

2.4.1 Why CouchDB

Couch DB is a Document DB specifically optimized for reliability and not speed. But it's still faster than the classical SQL database. It allows easy concurrency management and version control. Its main advantage is the scalability; a new server or database can be added to increase the storage or the compute power without any downtime or difference for the applications.

2.4.2 The storage

CouchDB use a Btree in which each document is stored in a block. B-Tree is a sorted data structure that allows for searching, insertions and deletion in logarithmic time. Research in it is O(log N) time, and the range is O(log N + K) In this database, nothing is deleted, this allows a revision history. This means that a deleted document will only be marked as deleted but still be on the disk. The database administrator needs to manually remove old document to avoid too much disk usage. To do so, he creates a new empty node and copy in it only the last document revision and replace the old node by the new. This allows the administrator to choose which version to keep but require more work to manage the DB.

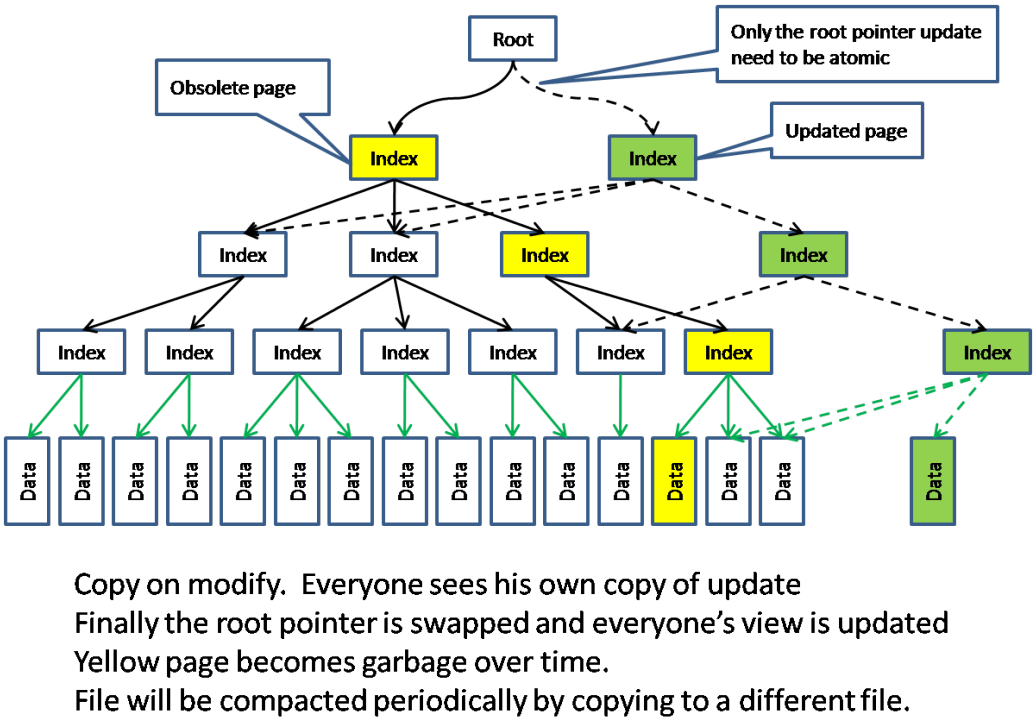

Figure 2.6: Btree storage [14]

2.4.3 concurrency

CouchDB allows better management of concurrency access than a SQL database. Due to the self containing nature of document, a change of data only happens in a single document. This allows

multiple access even when each access will modify the data without locking all the database like in SQL databases. A read access will just receive the version that is available, sometimes not the last one as seen on the figure 2.7. This allows CouchDB to run effectively under high load.

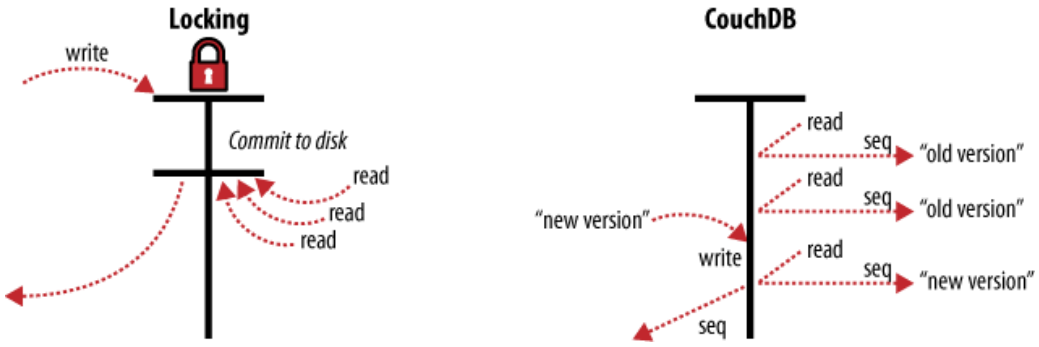

Figure 2.7: Concurrency in a document store: http://guide.couchdb.org/draft/consistency.html#figure/3

2.4.4 Managing the database

- Management center
  Couch DB can be managed from HTTP-based REST API or from a web interface called Futon. This section will be about the HTTP request that create and manage a simple database. We suppose that a CouchDB server is installed and run locally. After a command, CouchDB will return a JSON answer.

  The Futon web interface allows every command possible in a user friendly way and is ideal for simple management command.

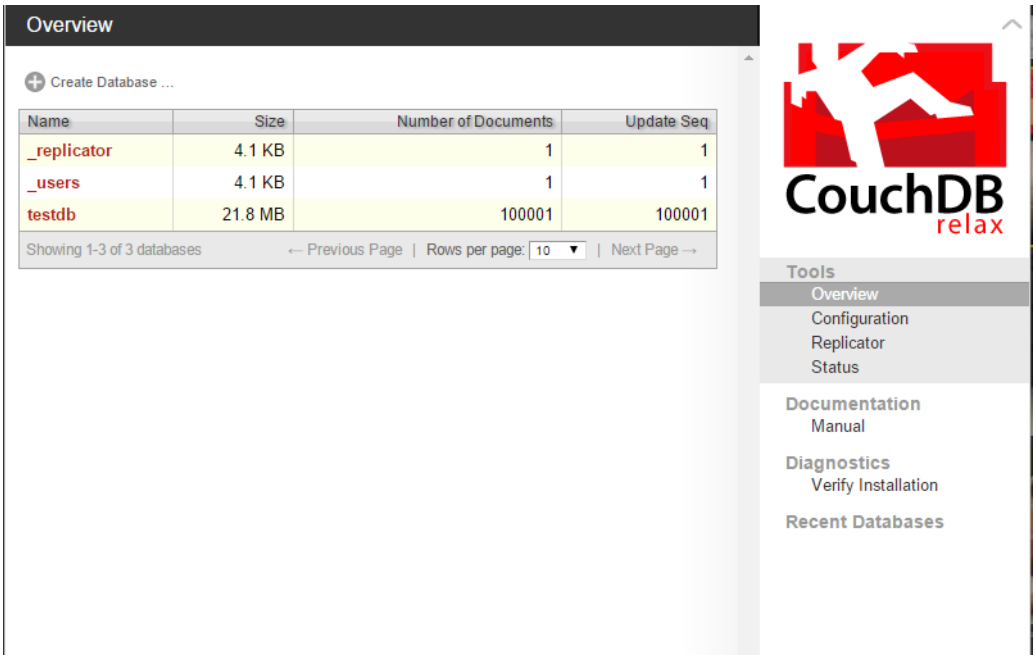

Figure 2.8: Futon interface

- Creating a database
  To create a new database, send a PUT request using curl for example:

curl −X PUT http://localhost:5984/new_database. The name of the database can only contain lowercase, number and/or underscore. the answer should be:

```
{"ok":true}
```

- Removing a database
  The request: $curl\ -X\ DELETE\ http://localhost:5984/database\_name$
  The answer:

```
{"ok":true}
```

- Adding data
  To add documents to the db use the following command and replacing the JSONFile with the file containing the documents in the correct JSON format.
  $curl\ -H\ "Content-Type:\ application/json"\ -d\ @JSONFile.json\ -vX\ POST\ http://127.0.0.1:5984/database_name/\_bulk\_docs$. The answer will be the list of the id of the documents added:

```
[{
    "ok":true,
    "id":"5631130c01d2d7c497622e3f",
    "rev":"1-5a0872dd28f5310aa61f2e6f8a1baf06"
}]
```

- Updating data
  The modification of data in CouchDB is always done by adding a new document with the same id but a different revision number. (See "Adding data" above).

- Removing Data
  They are multiple way to remove a document from the database; The first one involve adding a new document with the same name like in a modification but with the special field : "*_deleted*" : *true* in the JSON document. *dz* the second one invoke the HTTP DELETE request to remove it. The last one is putting the document in the "*_purge*" API. They are not the same, the first one will preserve all the field in the document but it will still be considered removed. The HTTP delete request will remove all the field in the document but it will still be there. Finaly the last one is not the correct one because the document will still be on disk but will be inaccessible [15].

- Versioning
  If you want to change a document in CouchDB,you load the full document out of CouchDB, make your changes in the JSON and save the entire new version of that document back into CouchDB. This means that every change are written on the disk in a new document and that every version of the document can be retrieved unless the database administrator cleaned the node of the document. To see them, send the revision number needed for the document in the HTTP request. Sometimes two document can be modified at the same time and will both be

marked as the newest one. When it happens, the database will detect it and notify with the *_conflict* flag, mark one of them as the new one and keep the other as a previous revision. It is up to the application that created the conflict to solve it.

- Replication
  In CouchDB, the replication between database is made incrementally to reduce bandwidth and processing time allocated to the replication. Replication is necessary to keep data consistency between two node of the DB or with the backup server. A database will trigger a replication between two server. Then the servers compare their data then the first one send the change. A replication is always uni-directionnal but a server that received change can also start a replication. This process can be propagated between multiple server (see figure 2.9

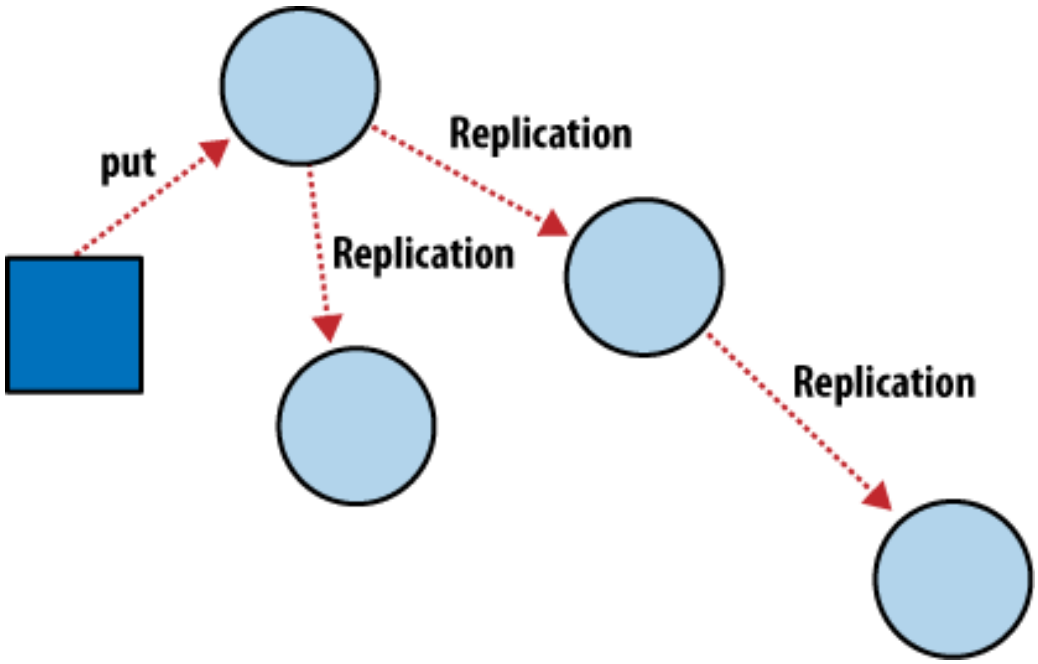

Figure 2.9: Replication in CouchDB: http://docs.couchdb.org/en/1.6.1/_images/intro-consistency-04.png

A replication can be local or remote. A local replication is between two databases running on the same CouchDB server. This is useful for local backup, or snapshots. To replicate locally use the following command:

```
curl -X PUT http://localhost:5984/_replicate -d
'{"source":"sourceDB","target":"destDB"}'
```

A remote replication is when the target is a database running on another server. This is useful for data consistency between multiple instance of a server. To replicate remotely use the following command:

```
curl -X PUT http://localhost:5984/_replicate -d
'{"source":"sourceDB","target":"http://targetadress.com/destDB"}'
```

A remote duplication can also have a local destination but a remote source, this is useful for data consistency because a replication is one sided.
Both source and destination can also be remote.

2.4.5 Querying the database

Couch DB use the map reduce paradigm as seen before. Here is an example of a map function that can be used on the data loaded:

```
function(doc){
    if(doc.type == 'buyer' && doc.wishes['fruits'].lenght > 0){
        emit(doc.gender,1);
    }
}
```

Here is an example of the corresponding reduce:

```
function(keys, values){
    return sum(values);
}
```

This is a very simple example that just query the gender of all the potential buyer of fruits and then compute the sum for each gender. Incremental Computation of Map Results

2.4.6 Specificity of Couch DB

- Design Document
  Design documents are a special type of CouchDB document that contains application code. They are created by adding *_design/* at the beginning of their ID. They act like other document for the database but will contains special field that will contains map/reduce code for the corresponding data or instruction on how to transform the data into a HTML or XML view. They can also include other file like for example CSS or image as attachment. The design document paradigm allows web-based application to be deployed and updated by a simple database replication with an easy rollback in case of problem. It also force the developer to use safe data transformation by using validated data request. The Design document API are complex and new field are added each major CouchDB update so this section is a simple overview and not a complete guide.

```
{
  "_id" : "_design/example",
  "views" : {
    "foo" : {
      "map" : "function(doc){ emit(doc._id, doc._rev)}"
    }
  }
}
```

Figure 2.10: A simple design document: http://guide.couchdb.org/draft/design.html#basic

- Validation
  Validation is a tool that allows the database to only accept correctly formed document. When the application will insert a erroneous document, CouchDB will return a error code 403 and the database won't be modified. To implements these validation, the user has to add a special function to the corresponding design document; the *validate_doc_update* Javascript function that will be called for each document added or updated. A design document can contains only one validate function but the database can contains multiple design documents with each their own validation. In this case, each function will be called on each document in an unspecified order (it can change for each request).
  This function accepts 3 arguments;

```
function(newDoc, savedDoc, userCtx){
    ...
}
```

  - *newDoc* is the document being added or the new version of the document.
  - *savedDoc* is the last revision of the document currently in the database in case it exists.
  - *userCtx* is the user that made the request and their roles

  The body of this function can include code that will make some test and return an exception if they fail. A validate function can never modify the documents on which they work, they needs to only read them.

  the two errors that the *validate_doc_update* function can throw are;

  - `throw({unauthorized: 'Error message here.'});`

    thrown when a user is not authorized to make the change but may re-authenticate.

  - `throw({forbidden: 'Error message here.'});`

    thrown when the change is not allowed.

  See figure 2.11 below for an example of a validate function.

```
function(newDoc, oldDoc, userCtx) {
    if (newDoc._deleted === true) {
        // allow deletes by admins and matching users
        // without checking the other fields
        if ((userCtx.roles.indexOf('_admin') !== -1) ||
            (userCtx.name == oldDoc.name)) {
            return;
        } else {
            throw({forbidden:
                    'Only admins may delete other user docs.'});
        }
    }

    if ((oldDoc && oldDoc.type !== 'user') || newDoc.type !== 'user') {
        throw({forbidden:'doc.type must be user'});
    } // we only allow user docs for now

    if (!newDoc.name) {
        throw({forbidden:'doc.name is required'});
    }

    if (!newDoc.roles) {
        throw({forbidden:'doc.roles must exist'});
    }

    if (!isArray(newDoc.roles)) {
        throw({forbidden:'doc.roles must be an array'});
    }

    if (newDoc._id !== ('org.couchdb.user:' + newDoc.name)) {
        throw({
            forbidden:'Doc ID must be of the form org.couchdb.user:name'
        });
    }

    if (oldDoc) { // validate all updates
        if (oldDoc.name !== newDoc.name) {
            throw({forbidden:'Usernames can not be changed.'});
        }
    }

    if (newDoc.password_sha && !newDoc.salt) {
        throw({
            forbidden:'Users with password_sha must have a salt.' +
                'See /_utils/script/couch.js for example code.'
        });
    }
}
```

Figure 2.11: A simple validate function: http://docs.couchdb.org/en/1.6.1/couchapp/ddocs.html

Chapter 3

Implementation

3.1 Implementation details

The following chapter will present briefly a test application made in Java using CouchDB to test and display all the specificity of a document database.
The application is a simple graphical user interface that manage display and manage a simple database. This database contains information about a Fruits and vegetables store and their clients. This application is coded in Java using the MVC principle and the lightcouch framework for the communication to the database. This framework is a simple wrapper over the HTTP request.

When the application launch, it will first display a pop up window that ask information about the database; the server, the name of the database , the password and the username. It will then store them in a Database object and reuse them for every request (see figure 3.2). Contrary to a SQL database, there is no need to open a connection , the server is not contacted until the first request.

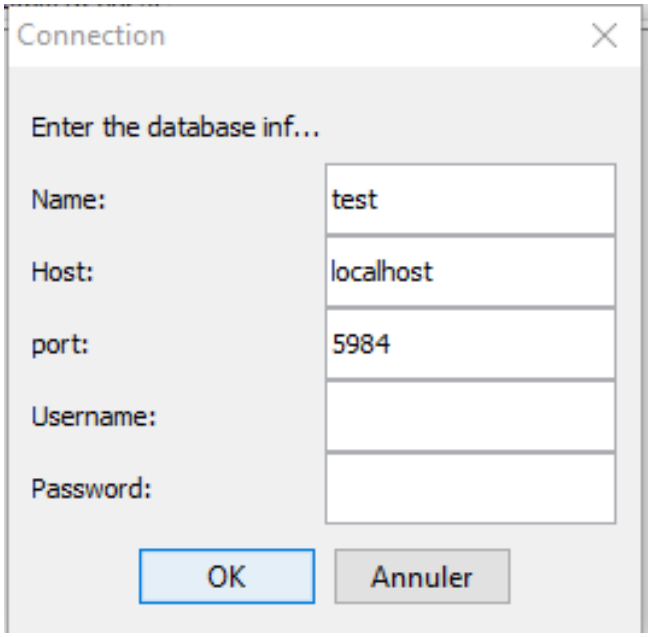

Figure 3.1: Connection Screen

```
CouchDbProperties properties = new CouchDbProperties(
                        name,
                        true,
                        "http",
                        host,
                        port,
                        username,
                        password
                );
CouchDbClient db = new CouchDbClient(properties);
```

Figure 3.2: Database connection object

after the connection, the main window will display a list of all the document in the database and basic information about the query that retrieved them.

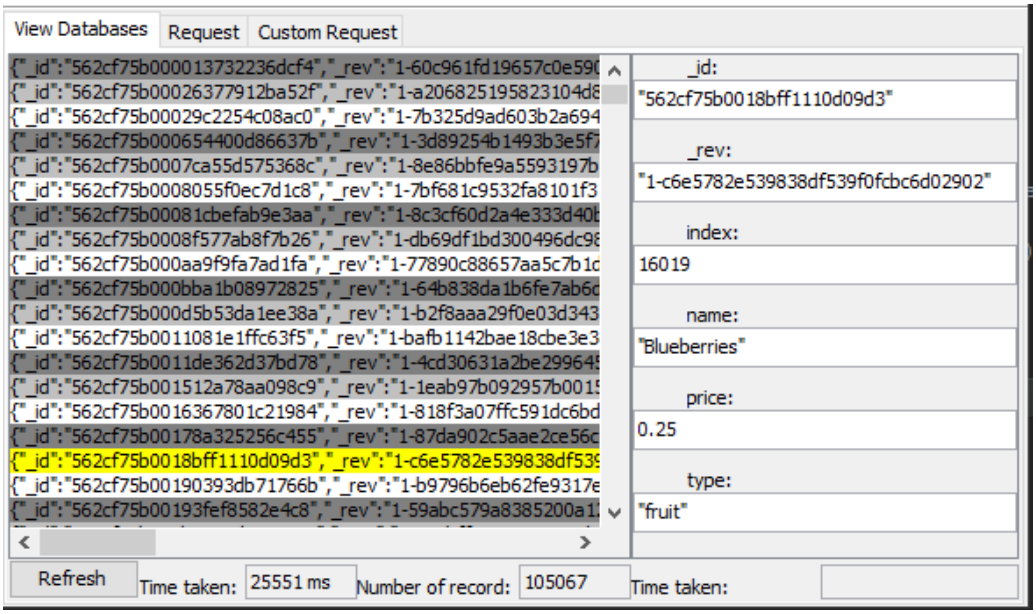

Figure 3.3: View window

It use an HTTP request to retrieve all the document and their value (see figure 3.4) The first line send a request to the view called *_all_docs* that CouchDB create by default for all database and that contains a list of all the documents in it. the parameter *.includeDocs(true)* specifies that the request needs the data of every documents and not just the keys.

```
List<JsonObject> allDocs = DBManager.getDb()
                                    .view("_all_docs")
                                    .includeDocs(true)
                                    .query(JsonObject.class);
System.out.println(allDocs);
```

Figure 3.4: Retrieve all the doc using Java

When a row is selected, more informations is displayed on the left of the application and allows deletion and edit of this specific document. The change are made by a simple request and the server answer "Ok" or with the new revision in case of an edit.

An other tab allows the user to select a predefined list of request. Those request use the principle of design document and are self contained in the database. Or if the request is not yet in the database, the user can write it and execute it in the lower part of this tab.
The map/reduce request must be written in Javascript with the reduce function and the map function separated in their corresponding area.
The application will then send them to the database by creating a new design document, putting the map reduce in it and adding it to the database. The application then call it from the design document and then when the result is return, it delete this design document.

This is due to the design of CouchDB that only allows map/reduce to be run from within the database and not from a simple request. It's a major design flaw in the application because that mean that every time the user will send a new request it will increase the size of the database due to the nature of the storage of CouchDB.
In a real world application, all the map/reduce will be already in the database. That allows the application to always use the last version without updating it and reducing the bandwidth by not needing to transmit all the request each time.

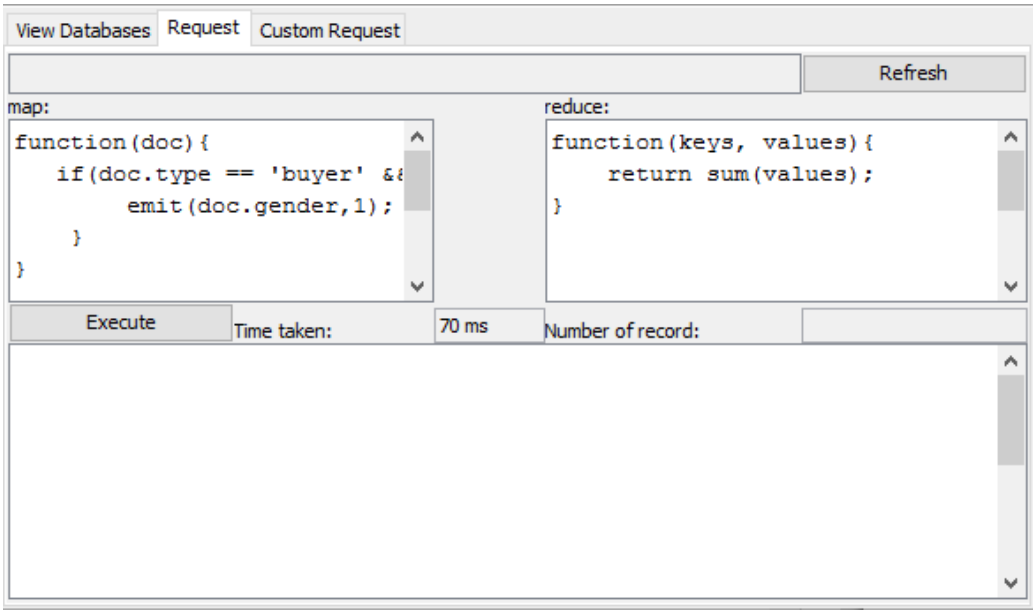

Figure 3.5: Request window

The last tab allows custom request to be sent to the database by specifying manually the parameters. The application will then append the user input with the address of the database and send it to CouchDB; $htttp://database\_adress.com/database\_name/custom\_input$

The result will be displayed in the lower frame as plain text. This allows all the other request not implemented in the application to be still executed.

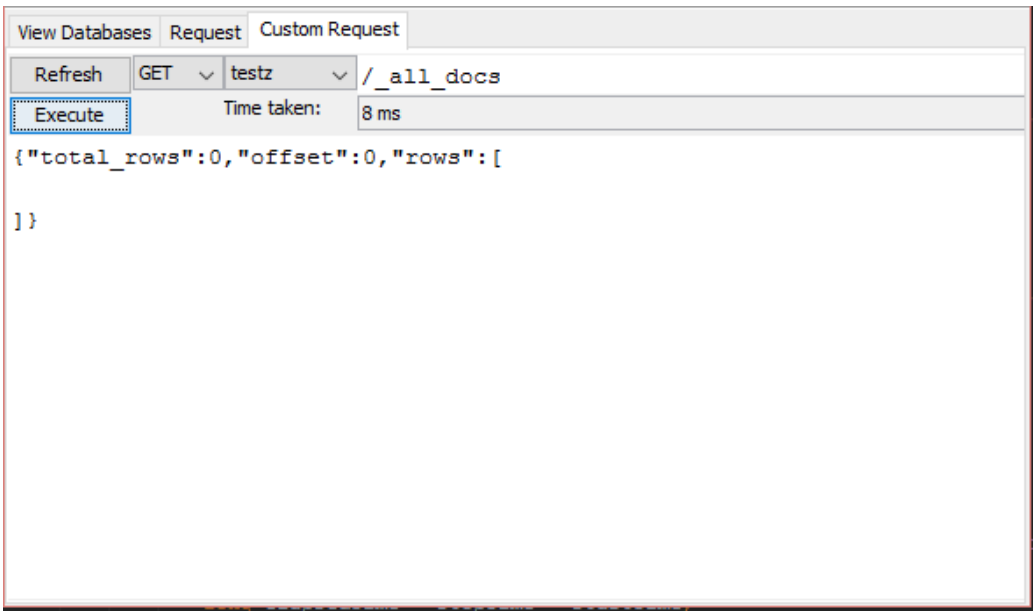

Figure 3.6: Result of a map/reduce

The application allows more command in a menu bar over the main window. This menu bar allows the creation of a new database, the deletion of the current database, the import of documents in bulk in the JSON format and the connection to a new database. All those requests will require the user to have the corresponding right over the database

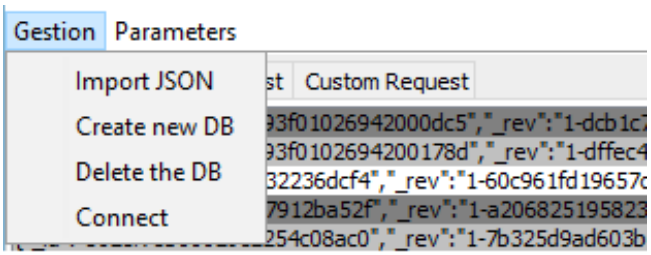

Figure 3.7: Menu bar of the application

Chapter 4

Conclusion

This document is an overview and a real world test of CouchDB. The conclusion of this is that a document store is a lightweight fast and reliable database optimised for the web and high load application with its lock free multi access. The database installation and maintenance is really straightforward and simple and the database request are plain HTTP request that require a few lines of code.

But even with all those advantages, CouchDB is not always the best document store.MongoDB is recommended in application that don't need the consistency and the history feature of Couch but benefice more of the speed advantage of MongoDB. This shows that even when a No-SQL database is better, CouchDB could not be the recommended one to use

When the application needs complex request like join or relation, SQL is still more appropriate. Document store also doesn't guaranty that an application will always access the last version of the data. Because of those, it's not destined to replace SQL but rather to complement it.

Bibliography

[1] Why NoSQL?. Retrieved from http://www.couchbase.com/nosql-resources/what-is-no-sql

[2] Tim Perdue. NoSQL: An Overview of NoSQL Databases. Retrieved from http://newtech.about.com/od/databasemanagement/a/Nosql.htm

[3] NoSQL. Last modified 2015, 7 december. Retrieved from https://en.wikipedia.org/wiki/NoSQL

[4] What is NoSQL? Retrieved from https://www.mongodb.com/nosql-explained

[5] Amine Chaghal. NoSQL : 5 minutes pour comprendre. (2011, February 14). Retrieved from http://blog.neoxia.com/nosql-5-minutes-pour-comprendre/

[6] Matthew Hanlon. A Case Study for NoSQL Applications and Performance Benefits: CouchDB vs. Postgres. (2015, December 2) .Retrieved from http://figshare.com/articles/A_Case_Study_for_NoSQL_Applications_and_Performance_Benefits_CouchDB_vs_Postgres/787733

[7] The Definition of Database Performance. (2011, June 3). Retrieved from https://datatechnologytoday.wordpress.com/2011/06/03/the-definition-of-database-performance/

[8] Scalability. Last modified: 2015, November 13. Retrieved from https://en.wikipedia.org/wiki/Scalability

[9] Mike Dirolf.Inside MongoDB: the Internals of an Open-Source Database.(2010, Mai 27). Retrieved from http://www.slideshare.net/mdirolf/inside-mongodb-the-internals-of-an-opensource-database

[10] Pramod Sadalage. NoSQL Databases: An Overview. (2014, October 1). Retrieved from https://www.thoughtworks.com/insights/blog/nosql-databases-overview

[11] Riyad Kalla. Understanding the CouchDB file format. (2011, December 20). Retrieved from http://www.mail-archive.com/dev@couchdb.apache.org/msg18257.html

[12] More CouchDB reading btree. (2008, September 24). Retrieved from http://ayende.com/blog/3608/more-couchdb-reading-btree-lookup

[13] Rob Ashton. RavenDB & CouchDB - Map and Reduce. (2010, June 6). Retrieved from http://codeofrob.com/entries/ravendb-&-couchdb---map-and-reduce.html

[14] Ricky Ho. NOSQL Patterns. (2009, November 15). Retrieved from http://horicky.blogspot.be/2009/11/nosql-patterns.html

[15] Nathan Vander Wilt. The three ways to remove a document from CouchDB. (2012, November 17). Retrieved from http://n.exts.ch/2012/11/baleting

[16] J. Chris Anderson, Jan Lehnardt and Noah Slater. Retrieved from http://guide.couchdb.org/draft/index.html