REPORT

DOCUMENT STORES AND COUCHBASE

December 20, 2017

George Kagramanyan
Léni Poliseno

INFO-H-415: Advanced Databases

Université Libre de Bruxelles
2017 - 2018

Contents

1 Introduction

A database is simply a collection of information. This information, which can be of various nature and storage structure, is managed via a database management system (abbreviated DBMS). The apparition of this technology was a big advance in the field of highly scalable data storage and retrieval. Since the size of database systems can be very large (e.g. banking systems, national registers or YouTube platform), it is important to allow a rapid access and efficient manipulation of its contents. The main objectives of this work are to understand the usefulness of document-oriented databases, compare the today's most relevant DBMSs in this area and finally analyze one particular software called Couchbase[1] to determine in which situations it can be beneficial to turn towards this solution.

1.1 Relational databases and SQL

This section briefly introduces the first widely adopted model and which remains very used in practice today: relational databases. It is a more powerful alternative compared to its predecessors hierarchical and network database models that were inflexible and not optimized for the INSERT operation [1].

Relational databases were designed to keep the information coherent and consistent at all times while allowing multiple persons to work simultaneously on same data. That is, they are following the ACID concept [2]. In this new setting, each piece of information is represented in form of a tuple and is stored by rows into tables (also called relations). Although, the internal structure of tables with their interconnections might be complex, it is relatively easy to manage stored information by means of a high level programming language: **S**tructured **Q**uery **L**anguage. Initially, the objective was to create a simplistic language that can be understood by users that do not necessary have programming skills. Nowadays, SQL and all its variants can be seen as a standard methodology used by relational database management systems (RDBMS).

[1] Official website https://www.couchbase.com/

2 NoSQL

"NoSQL is a set of concepts that allows the rapid and efficient processing of data sets with a focus on performance, reliability, and agility." is a definition given by [3].

It might be interesting, however, first to understand why the traditional RDBMS were no longer able to fulfill company's needs which gave birth to search for other means of data storage and consequently to NoSQL. As a direct consequence of Moore's Law, processor's computation speeds did not improved much from around 2005. Therefore, the performance of RDBMS could not be resolved anymore by acquiring faster processors. On the other hand, as illustrated on Figure 2.1, the size of datasets is increasing exponentially due in some part to a popularization of web, mobile and Internet of Things applications. Most of those applications need to handle a big number of concurrent users, be available 24h/24h, use various formats of data and quickly adapt to new requirements to satisfy the customers. This required big enterprises to reconsider their management policy about how they deal with and store those massive amounts of information.

The solution was then to adopt parallel processing where the workload was distributed among multiple processors. First initiative was done by Google's society who needed to index billions of web pages for search queries. In 2004 they revealed MapReduce framework that allows to distribute computations on computer clusters which results into highly scalable and parallel system. Along with other software innovations, this was one of the most important fundamentals for the NoSQL technology. Because, as will be mentioned below, scaling to multiple computers is more challenging for RDBMS.

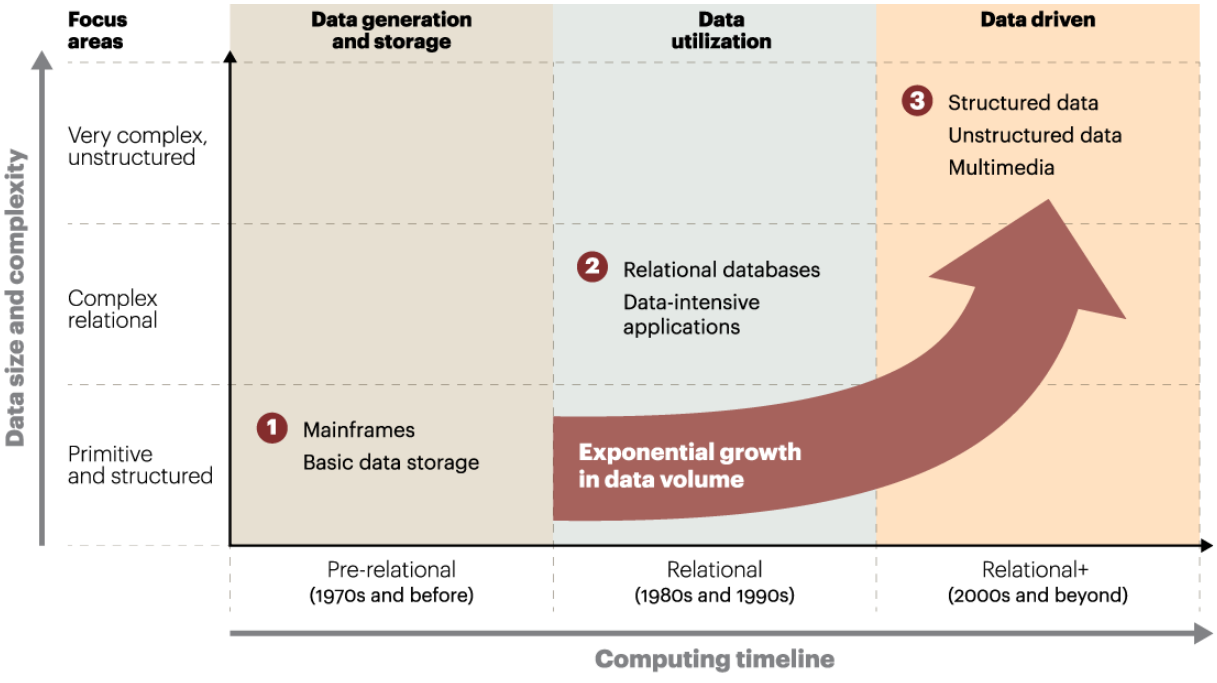

Figure 2.1: The evolution of Big Data. Image taken from [4].

Returning to the definition, we can say that NoSQL is a way of managing data in a more open environment (i.e. there is no need for an entity-relational model). Big Data is not the only reason where this model is preferred over RDBMS, variability (for example adding only one extra attribute to one entry is impossible without adding a column for the whole table in relational model, which can be seen as a waste of resources) and agility (requests time should be minimum) are also points that must be considered. There exist many different types of data stores available for NoSQL applications. The four main categories are: document stores, column stores, key-value stores and graph stores. Additionally, it should be noted that NoSQL databases also use SQL-based query languages.

2.1 BASE transactions

Like we said before, relational models follow ACID transaction rules. In other words, they require that any change in tables (concurrent or not) leaves them consistent and preserves data integrity. A typical example where this concept applies are bank transactions - either the whole operation succeeds and money is transferred to another account or one of its stages fails which results to abort the mission and return to initial state.

The reason of why NoSQL databases usually do not guarantee consistency at all times is illustrated by the following example. Today, for the questions of performances, some databases are distributed around the world, which means that each node maintains a copy that is continuously synchronized. When someone posts a comment on a YouTube video, a RDBMS would ensure that each database copy is updated before anyone could see the written text. This can create an unnecessary latency in the sens that if one person sees the comment a little bit earlier than someone on another side of the planet, it is not a big deal. NoSQL tolerates some delays in the changes and assures *eventual consistency* which forms a part of BASE transaction system [5, 6]. This stands for :

- Basic Availability : the data and services should be available even in the case of some failures of requests or of the nodes holding a replicated database.
- Soft-state : temporal inconsistency and inaccuracy of information is allowed.
- Eventual consistency : sooner or later the system must reach its consistent state. Contrary to ACID model, there is no guarantee about when this moment will occur.

To conclude this section, ACID transactions focus on data safety while in BASE, systems availability is put on the first place. Not all NoSQL databases use BASE proprieties. The choice between these two transaction types is also in big part dependent on the applications needs.

2.2 RDBMS and NoSQL recapitulation

For the sake of a better understanding and comparison, we decided to list some strong/weak points and characteristics of the two systems.

Strong points of RDBMS:

- Based on a standard query language SQL which is portable to various databases.
- Big community of experienced users.

- Assures data integrity through ACID transactions.

Weak points of RDBMS:

- Scale well only on a single server. Spreading the load to multiple nodes increases complexity.
- Lots of JOIN operations may decrease performance.
- Problematic to work with semi-structured data.
- Object-relational mapping layer can become complex.
- Schemas must be defined prior to adding data. This does not promote the agile development approach [8].
- Failure of the main server compromises the entire database.

Strong points of NoSQL:

- Better suited for scaling across a big number of processors and scales linearly by augmenting the number of nodes.
- More flexible since it can handle structured, unstructured and semi-structured data. Does not require a schema.
- Object-relational mapping layer is not needed (see Figure 2.2).
- Supports powerful full-text search libraries such as Lucene.
- JOIN operations can be avoided or minimized.
- More open source solutions.

Weak points of NoSQL:

- Databases usually have a proprietary nonstandard query language that limits portability and migration.
- Younger than RDBMS which means less experienced users and weaker community support.

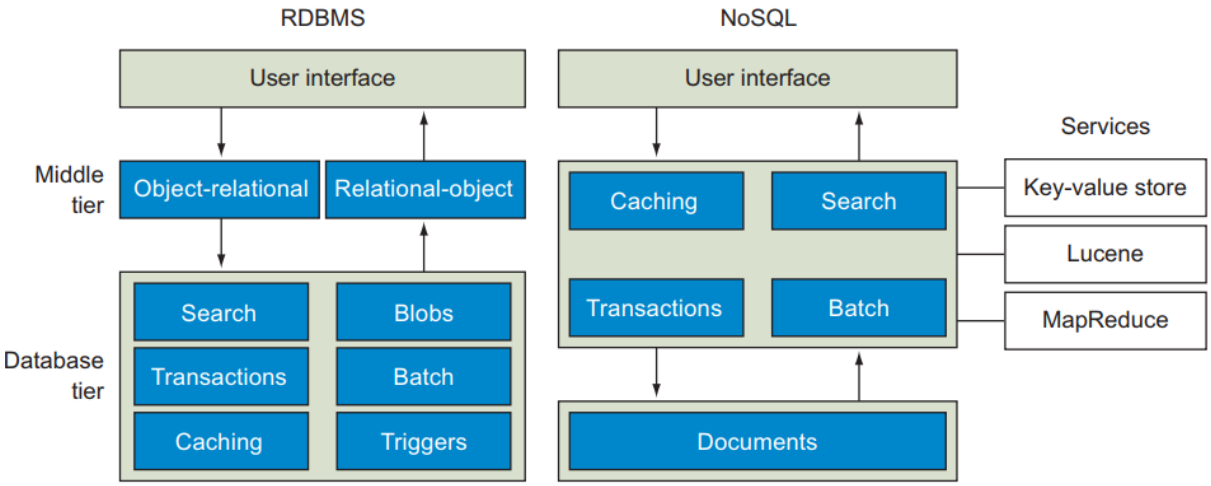

Figure 2.2: Representation of different application layers. RDBMS have integrated many methods in the database layer allowing them to have a total control over the operations and guarantee security and integrity of manipulated information. While NoSQL models have relaxed their databases by moving functionalities into the middle layer. Image taken from [3].

3 Document stores

Although many types of NoSQL databases exist, from now on we will concentrate our attention on document stores. The latter can be defined as a way of representing (semi)structured information in a hierarchical, tree-like structure. Where the data objects are usually stored at the end of branches of such tree in a form of documents. Each branch is associated with a path expression allowing to navigate between nodes. To make the navigations and data retrieval easier, documents can be grouped together into collections. Like the hierarchical file systems used in various OS, collections provide a logical grouping of similar documents. Moreover, representing relations, as in relational databases, between documents is usually discouraged (but could also have advantages as we will see it later) since all the associated data with a record can be stored in the document itself [9, 11].

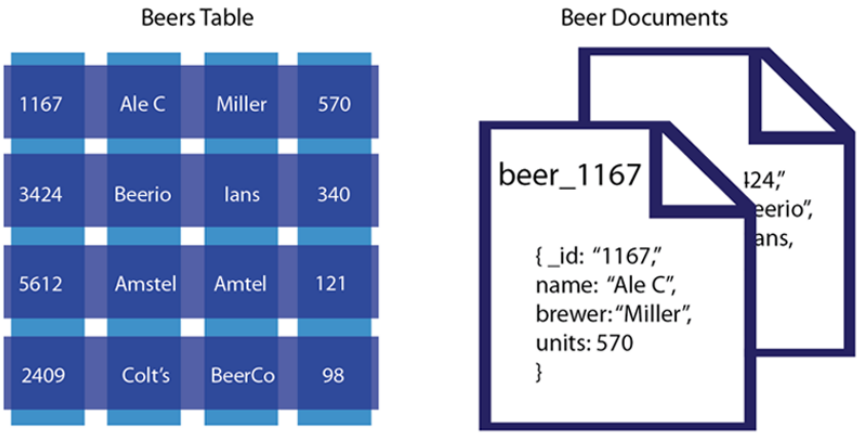

Figure 3.1: Comparison of data representation in a relational model (left) and document stores (right). Image taken from [12].

The major difference with a key-value store[1] is that in document stores, every added document is automatically indexed within the database. It means that the whole content of the document becomes searchable allowing one to quickly extract any required part of several documents at once. While key-value stores are not searchable and will always return an entire value (or document) [7]. This also means that a document store can easily simulate a key-value store.

In sum, document stores offer many appealing possibilities for storage of data in various forms and types. They do not enforce a schema allowing two documents to have mutually exclusive optional values and thus reducing storage space for unnecessary arguments. Hence, this gives them a hight flexibility and freedom. Furthermore, since documents are

[1] Simple database, without a query language, that maps strings (unique keys) to arbitrary large pieces of data (values).

usually independent entities, it increases the performance for read and write operations and facilitates replication from one server to another. Agile methodologies allow the system to quickly adapt to the new data structures that could potentially submerge in future. Some of the current successful applications for document stores databases include: e-commerce, product catalogs, ratings and comments history, tweets and blog posts, in-game statistics, web analytics, data collected from embedded devices and many more.

3.1 Internal data structure

Until now, we haven't specifically defined the notion of a *document*. It is the most common concept of all document-oriented databases which can be seen as a (semi)structured information encoded typically in XML[2] or JSON[3] (with its supersets) format [10]. Both use a human readable syntax roughly consisting of a description of the data and its associated value. Below is an example of an XML document :

```
<belgian_cuisine>
       <food>
              <name>Strawberry Belgian Waffles</name>
              <price>7.95$</price>
              <description>
                Light Belgian waffles covered with strawberries and whipped cream
              </description>
              <calories>900</calories>
       </food>
</belgian_cuisine>
```

And its copy in JSON format:

```
{
    "belgian_cuisine": {
        "food": {
            "name": "Strawberry Belgian Waffles",
            "price": "7.95$",
            "description": "Light Belgian waffles covered with strawberries and
                            whipped cream",
            "calories": "900"
        }
    }
}
```

JSON is often considered to be a better alternative to XML [13] and a standard format for mobile, web and Internet of Things applications. Additionally, it allows simple interconnections between different documents such as 1-to-1, 1-to-many and many-to-many [14]. To illustrate this, we can rewrite the above example as a combination of two documents instead of one:

```
{
    "belgian_cuisine": {
        "food": {
            "name": "Strawberry Belgian Waffles",
            "details": "waffles_details"
        }
    }
}
```

[2] Extensible Markup Language.
[3] JavaScript Object Notation.

The second document is referenced via the *details* field:

```
{
    "waffles_details": { // assuming this is also the key
        "price": "7.95$",
        "description": "Light Belgian waffles covered with strawberries and whipped
                        cream",
        "calories": "900"
    }
}
```

In RDBMS, doing this is a common routine. While in document stores, the choice of referencing or not depends on the application. Here are some of the advantages of this practice:

- When documents are not frequently accessed together, each one can be independently sent over the network. Usage of cache memory and network's load decreases.
- In the case of a 1-to-many relationship, if an update is required on the "1" side, this is easily done without touching the "many" side. But, if it was only one document, we then had to search the whole document for each item from the "many" part and update the information that it uses from the "1" part. Which can decrease efficiency.
- As for RDBMS, referencing can prevent repetition of attributes and thus requires less storage.

There are also disadvantages of referencing :

- When connected documents are often used together additional complexity and memory operations may be added.
- Many document stores provide ACID properties on document level, meaning that we can never start manipulating partially-updated documents. Therefore, two distinct documents, referencing one another and that must to be updated, can not take advantage of a single consistent operation.
- Not all document stores support JOIN operation.

3.2 Today's most used document stores

As of October 11, the ranking table showed in [15] contains a classification of a total of 43 different document stores and multi-models. Systems are ranked by their popularity and the score is computed based on criteria such as number of results related to the system in search engines, frequency of searches in Google Trends and discussions appearing in question-answer sites such as Stack Overflow, number of job offers mentioning the system and others. We thought it could be interesting to look closer to the top-5 systems from that list. Since Couchbase occupies position number 3 and will be discussed in the next chapter, we selected the first five other systems with highest scores. In decreasing order :

- MongoDB : Open-source, distributed and powerful document database that stores data in a JSON-like syntax. It is not the oldest one but has a bigger community compared to its rivals. Has a rich query language, tunable consistency and supports operations on geospatial data. Provides official and non-official API's written in dozens of languages [16, 17].

- Amazon DynamoDB : Developed at the Amazon company, is a fully managed cloud NoSQL database that, among others, offers a service to manage triggers. Usually combined with Amazon DynamoDB Accelerator (a specifically designed caching service) that improves performance and drastically reduces response times. Documents can be in JSON, HTML or XML encoding. However, this technology is not free [18].

- CouchDB : Open-source project under Apache License that is more oriented into mobile phones and web applications since it uses a native JSON format for storage. Has an intuitive yet powerful query engine and the whole architecture is designed to be fault-tolerant and in a controlled environment. With focus to maximally simplify the developers work [19].

- Azure Cosmos DB : Multi-model platform suited for document stores, key-value stores, column stores and graph stores. Works with JSON documents and supports the well known SQL query language. Databases have a particularity to be easy deployable in many regions around the world and more places are continuously added to the list [20]. Guarantees 99.99% availability for each region, zero-data losses after failures of some servers and has a tunable consistency on five levels (from strong to eventual) that allows clients to choose optimum trade-offs between consistency, availability, latency, and throughput. Not free but claims to be cheaper than Amazon DynamoDB [21].

- MarkLogic : Also a commercial, multi-model NoSQL database that provides storage for XML, JSON and RDF documents. Also supports SQL and similar query languages. Has ACID properties that can apply to multi-document transactions across a cluster. Can be deployed in a physical environment but can also run from a cloud provider including Amazon Web Services, Microsoft Azure and Google Cloud Platform. Has a bi-temporal feature allowing to differentiate between occurred (valid time) events and recorded (system time) events [22].

All these technologies support frameworks in multiple languages, all are highly scalable, all claim a high availability, low latency and all are evolving very quickly so the best place to gather information about them are the official websites.

4 Couchbase

Couchbase Server, originally developed in 2010, is a distributed and document-oriented database that stores data in JSON format. It is designed to deliver a scalable, powerful and highly available service. The platform can also be used as a key-value store and has a very similar to SQL query language “N1QL” which is optimized for documents having any structure written in JSON and that returns JSON as result. Couchbase is open source and is available in Enterprise Edition (paid version) and Community Edition (free version) which has less features and is advised for non-commercial developers [23]. In order to be able to easily increase or decrease the size of a cluster, the architecture consists of perfectly identical nodes (in the sens of installed software) that exchange messages with each other on demand. Additionally, this simplifies physical or distant configurations and troubleshootings of a cluster.

Replication and an even workload distribution across nodes in the cluster is an automatic process and the database size can rapidly grow simply by adding more hardware. This allows a straightforward and transparent administration and eases the work for developers. Besides replication, high availability is assured by excluding downtimes and keeping the system online even during various maintenances, upgrading operations and addition or removal of nodes. Moreover, fault tolerant mechanisms and cross-cluster replications[1] can almost guarantee an uninterrupted availability in case of unexpected events and node/cluster crashes.

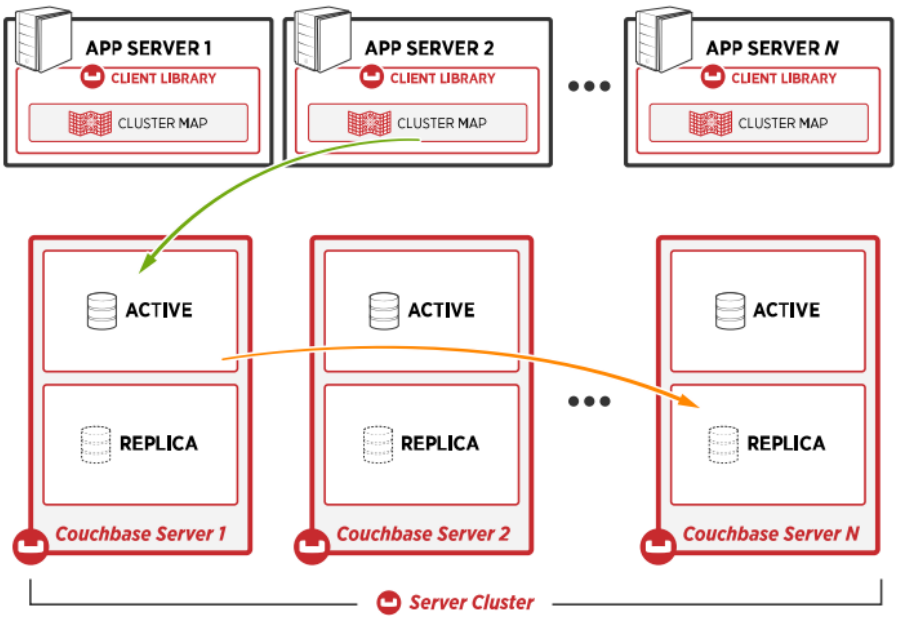

Figure 4.1: Replication is an important and asynchronous process during which one to three copies of active data are transferred to another node in the cluster. At any moment, only one copy of certain data is active and by default all the read/write operations are directed to it. If replication is enabled, an automatic mechanism ensures that data modifications are forwarded to the corresponding replicas in real time. Additionally, applications may specify how many replica copies they want to keep or to request a read of one of the replicas. Image taken from [28].

Couchbase Server can continuously handle heavy traffic from modern online and interactive applications with minimal latency. Furthermore, there exists a Couchbase Mobile platform which is simply an embedded database that runs on mobile applications (e.g. Android and iOS) with and possibly without a network connection [24]. It additionally helps customers to easily and securely synchronize data between multiple devices or the cloud and allows big companies to efficiently combine the two solutions.

4.1 Architecture

Each Couchbase node consists of same components and services (Figure 4.2), they are described below.

[1] Cross Data Center Replication (XDCR) is a technology designed to replicate the whole cluster into one or more other locations.

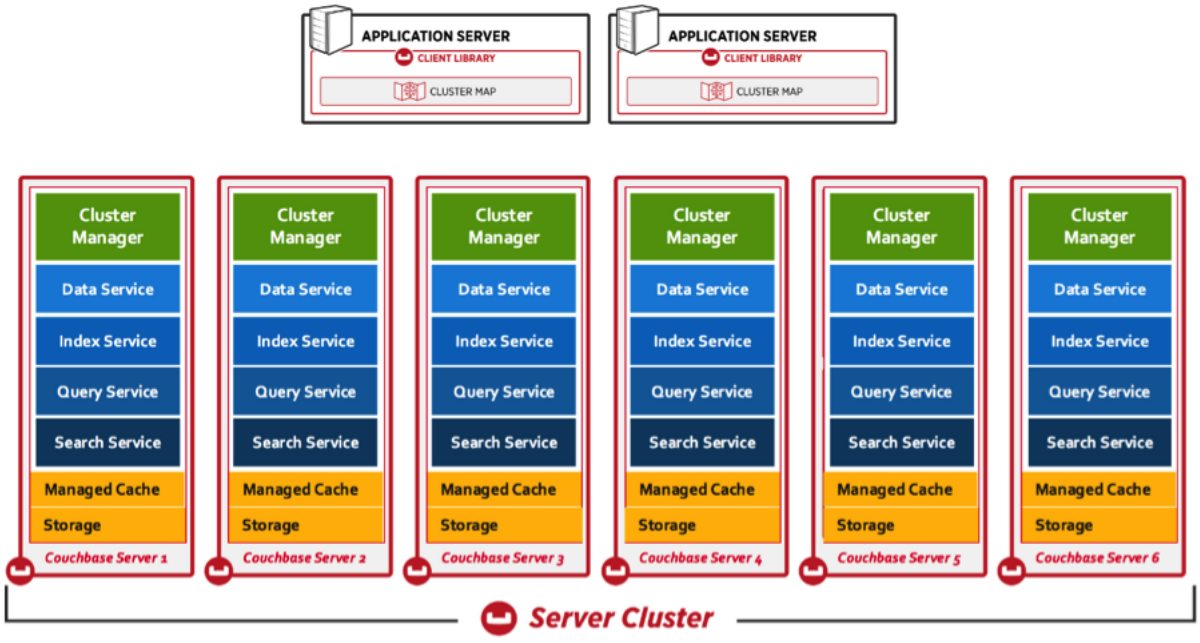

Figure 4.2: Architecture of nodes within a cluster. Image taken from [26].

4.1.1 Cluster Manager

Cluster level operations such as logging, statistics collection, connection authentication, cluster topology and data placement are all done by the Cluster Manager. Indeed, from time to time, cluster topology constantly changes due to addition and deletion of nodes. Therefore, nodes must be re-coordinated in order to ensure the optimal distribution of the load. The Cluster Manager takes care that all nodes in the new setting inherit the correct cluster configuration while keeping the system uninterrupted. This process is called *rebalancing*.

Another main task is failure detection of unresponsive nodes. Each Cluster Manager, of each node within a cluster, communicate with each other by sending signals at fixed intervals, called *heartbeats*, which simply contain some basic statistics about the node. Even if all copies of Cluster Manager are identical, there is always one node that is elected by others to become an *orchestrator*. The orchestrator maintains an authoritative copy of the cluster configuration to avoid conflicts and distributes its changes. By keeping track of the received heartbeats, it is essentially responsible of automatically re-directing traffic to the healthy nodes from the ones that became unresponsive (i.e. no heartbeats were received after a specified period of time) and activating the relevant replicas. This process is called *failover* and can also be done manually by the administrator.

4.1.2 Data Service

As its name suggests, Data Service provides documents access and management. CRUD (create/retrieve/update/delete) operations in particular, which make use of a key (or ID) associated with each document and their value encoded in binary or JSON format. Additionally, the Data Service maintains the MapReduce view indexes where it allows developers to define their proper map and reduce functions. Views allow to calculate the answer in advance which results into faster N1QL queries. Similarly, there are MapReduce spatial view indexes which are adapted for geographic data.

4.1.3 Index Service

Along with these MapReduce indexes, there are Global Secondary Indexes (GSI) that can also accelerate queries. GSI are defined by the N1QL query language on a specific subset of documents depending on the targeted subset of fields. Independently of the Data Service, GSI are partitioned across the nodes that have the Index Service enabled (more details about service enabling/disabling on Figure 4.3). In sum, the Indexing Service plays a role of the index manager and interacts with the Query Service.

4.1.4 Query Service

The Query Service receives N1QL queries, parses them and tries to optimize before execution. To satisfy a request with an appropriate result, it can only be done of course with the help of the Data and Index services. Also, during the optimization phase, any existing keys defined by MapReduce views or GSI are automatically loaded in order to accelerate especially "GROUP BY", "ORDER BY", and "WHERE" operations.

4.1.5 Search Service

Search Service provides a Search API that allows to perform keyword searches on the data. It maintains full text indexes based on JSON documents and it is advised to be used instead of queries when the returned result depends on whether a certain field contains a certain string or not.

4.1.6 Storage

Different things that need to be stored have different storage requirements and Couchbase offers an optimized storage engine for each of its components. For example, each entry of GSI is represented as a ForestDB file in the file system. ForestDB is a special key-value storage engine with a B+trie structure developed by Couchbase. On the other hand, Couchstore engine, with a B+tree structure, assures the storage for the Data Service. Those two engines follow the append only principle. It means that all the incoming writes are added at the end of files and links to the previous versions are invalidated. A light compaction process needs periodically to take place in order to clean the fragmented space and produce a new contiguous file.

4.1.7 Cache

Couchbase is a memory-first database architecture. It moves the data to and from the disk by request but all the key-value operations are performed through in-memory cache for rapidity. Developers have even a possibility to store the documents in the memory and forbid their storage on disk, as shown in section 4.3. Data Service, Index and Search services and Query Service have three unique managed caching mechanisms optimized for their needs. Moreover, each service has a defined RAM quota indicating how much memory can be allocated for storage. Except the Query Service which manages its memory automatically.

The general aim is to find and keep the most frequently accessed items in the caching layer (RAM) to minimize the operations with the persistent storage (disk).

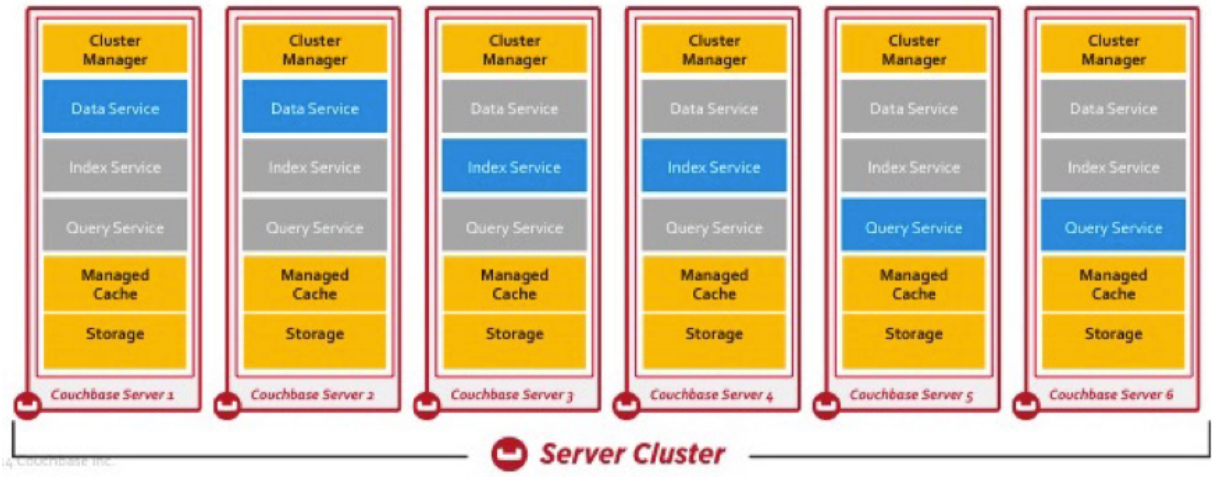

Figure 4.3: This picture shows that nodes can be dedicated to only run a subset of services (highlighted in blue) and disable others (in gray). It is called Multidimensional Scaling where the architecture can be modified on the software level in order to quickly address changing requirements by allowing each service to scale up and out independently from others. As opposed to Symmetric Scaling where each service within a node shares an equal part of the workload. Image taken from [26].

4.2 Installation

The easiest way to run a local instance of Couchbase Server is through Docker[2] software that offers an operating-system-level virtualization. Assuming Docker is installed, it then suffices to execute the following command :

```
docker run --rm -t --name db -h 127.0.0.1 -p 8091-8094:8091-8094 -p 11210:11210 \
couchbase/sandbox:5.0.0-beta
```

This is advantageous because it can be run from any OS. If the server's image exists, it will be loaded, if not, a new copy will be automatically downloaded. Once the command has finished, a by default configured server is up and ready to use. Here we use the latest beta version[3] which comes with several improvements [25] over previous stable 4.x versions. Although, it is still in development and some issues may exist, we suppose that it should be enough for our testings on a local single node.

We can now log-in and access via $http://localhost:8091$ to the *Couchbase Web Console* that offers a rich number of possibilities to the administrator. Starting from closely monitoring each aspect of any server to managing documents and executing queries.

[2] Can be downloaded from https://www.docker.com/get-docker

[3] At time of submission of this report, the official version was already released but it was late for us to switch.

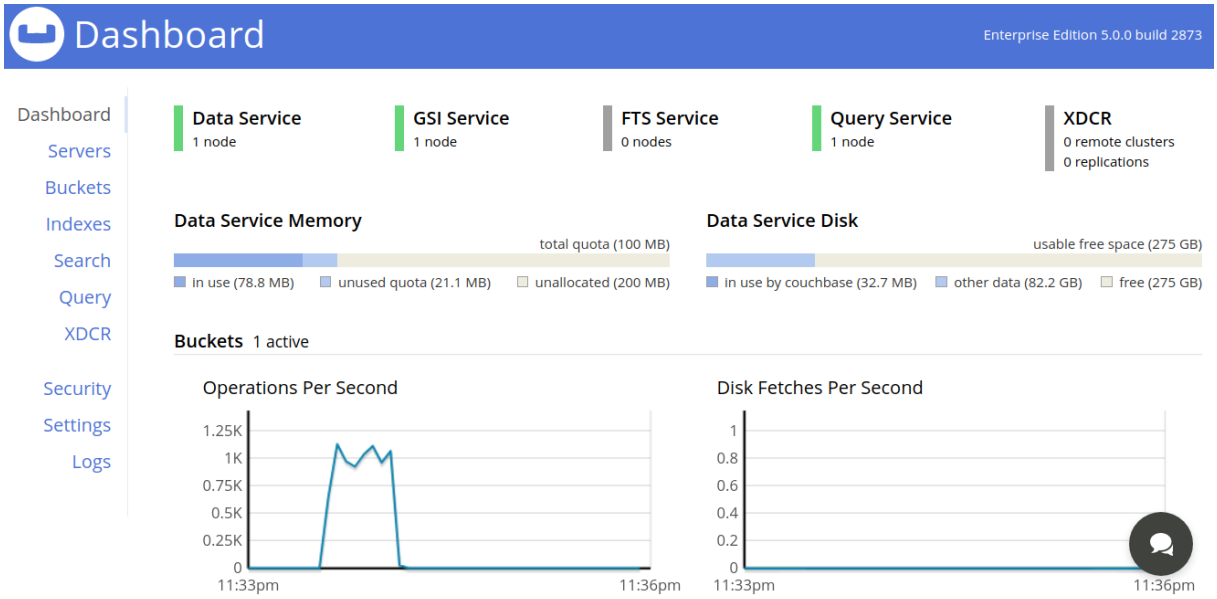

Figure 4.4: Couchbase Web Console showing the current state of the cluster.

One useful notion that can be defined at this stage is the notion of "buckets". The principle is analogous to collections meaning that buckets store logical groups of items (JSON documents) providing a centralized management (such as resource allocation and security properties) of items that they contain. Therefore, each bucket has its own and independent range of indexes.

4.3 Methods of administration

Besides Couchbase Web Console, monitoring and administrative tasks of clusters and servers can be done through command-line interface (CLI) and REST API. Since in this setting, Couchbase Server resides in a virtual environment, we do not have access to the CLI tool that comes together with a normal installation. However, same operations but maybe in a less intuitive manner can be done with the HTTP REST protocol[4]. The objective of this section is to illustrate a couple of things that can be done with the latter.

Retrieving buckets information :

```
curl -X GET -u Administrator:password localhost:8091/pools/default/buckets
```

The curl tool allows to communicate with servers using various protocols including HTTP. For each request to a Couchbase Server we need to provide a username and a password which is done with $-u$ argument. In this case, we ask information for every bucket that exists. Of course, specifying a particular bucket is also possible. The command output is in a JSON format :

```
  {
      "name": "travel-sample",
      "bucketType": "membase",
      "authType": "sasl",
      "proxyPort": 0,
      "uri": "/pools/default/buckets/travel-sample?bucket_uuid=1991a179c335ded1e56...
      ...
```

[4]More information about REST can be found here: https://en.wikipedia.org/wiki/Representational_state_transfer

```
        "replicaNumber": 1,
        "threadsNumber": 3,
        "quota": {
            "ram": 104857600,
            "rawRAM": 104857600
        },
        "basicStats": {
            "quotaPercentUsed": 77.21810150146484,
            "opsPerSec": 0,
            "diskFetches": 0,
            "itemCount": 31591,
            "diskUsed": 35286102,
            "dataUsed": 29539328,
            "memUsed": 80969048,
            "vbActiveNumNonResident": 10411
        },
        ...
```

The result is too long to show all information here. $travel-sample$ is the only bucket currently present on the server (default configuration) and we can remark for example that it contains 31591 documents.

Similarly, a new bucket can be created with :

```
curl -X POST -u Administrator:password -d name=newbucket -d ramQuotaMB=200 -d\
authType=none -d bucketType=couchbase http://localhost:8091/pools/default/buckets
```

This command can take various other parameters and returns nothing (not counting the HTTP response code) in case if the operation was successful. Alternatively, $bucketType$ argument can take the value "$memcached$" or "$ephemeral$". The former is designed to cache frequently used data and thereby reduce the amount of queries. The latter was introduced in this Couchbase's last version and allows to store all the data completely in RAM. This might be a faster alternative when applications require too much operations with disk, which is much slower compared to read/write operations executed in RAM.

Other administrative tasks include : software upgrading, cluster/nodes/buckets configuration, security settings, recovering and backup operations, cross data center replication and more. A complete list of REST API possible commands is described in [27].

4.4 Querying documents

REST API is also capable to execute N1QL queries. For example :

```
curl -u Administrator:password -v http://localhost:8093/query/service\
-d 'statement=SELECT name FROM `travel-sample` LIMIT 2'
```

This statement returns the value for a *name* field of the first two documents in the bucket as follows :

```
{
    "requestID": "9a4334d2-20d1-4529-891b-ff2b198ef8d4",
    "signature": {
        "name": "json"
    },
    "results": [
        {
            "name": "40-Mile Air"
        },
        {
            "name": "Texas Wings"
```

```
        }
    ],
    "status": "success",
    "metrics": {
        "elapsedTime": "33.118629ms",
        "executionTime": "32.998133ms",
        "resultCount": 2,
        "resultSize": 90
    }
}
```

Usually, developers who write code for their applications would prefer an easier way to perform read/write operations and queries on documents within clusters. For that purpose, Couchbase provides software development kit's (SDK's) supporting Java, Node.js, Python, Go, PHP, .NET and C languages. We are familiar with three of those and for no particular reason we have chosen Java SDK (version 2.5.1) to present some basic functionalities.

4.4.1 Accessing a bucket

After linking to the project all necessary libraries that can be found on the official website, we write the following code to initialize a connection with a cluster and then access to a particular bucket :

```
Cluster cluster = CouchbaseCluster.create("127.0.0.1");
cluster.authenticate("Administrator", "password");
Bucket bucket = cluster.openBucket("travel-sample");
```

For real world applications, it is recommended to pass a list of node's IP addresses to the *create*() method so that in case connection to the first address fails, next IP's can be tried. Also, it is possible to provide a password argument to *openBucket*() in case if it is protected. At the end, when all operations are done, we need to disconnect from cluster which will also automatically close opened buckets :

```
cluster.disconnect();
```

4.4.2 Creating documents

```
JsonObject library = JsonObject.create()
                        .put("name", "BSH")
                        .put("address", "Campus Solbosch")
                        .put("opening_hours", JsonArray.from("working_days 8h - 20h",
                                                             "saturday 10h - 17h"))
                        .put("characteristics", JsonObject.create()
                                                .put("floors", 8)
                                                .put("seats", 2198));

bucket.insert(JsonDocument.create("doc_1", 20, library)); // expires after 20 sec
System.out.println(bucket.get("doc_1").toString().substring(12));
```

After formatting the output for readability, here is what we obtain :

```
{
   "id": "doc_1",
   "cas": 1508502003163725824,
   "expiry": 0,
   "content": {
      "name": "BSH",
```

```
    "opening_hours": [
       "working_days 8h - 20h",
       "saturday 10h - 17h"
    ],
    "characteristics": {
       "seats": 2198,
       "floors": 8
    },
    "address": "Campus Solbosch"
  },
  "mutationToken": null
}
```

The *insert*() method must take two mandatory arguments : document ID ("doc_1" in this example) and the content (a *JsonDocument*). Note that other formats for the content are supported, such as *JsonLongDocument*, *RawJsonDocument*, *SerializableDocument* and *BinaryDocument* (which stores binary data). One of the optional parameters is the expiry time given in seconds (here 20). After the end of that period, the document becomes no longer accessible and will be deleted. By default, documents are kept forever (*expiry* = 0) and that is what we see in the output. This might be a little display bug of this beta version because by looking at this document from the Couchbase Web Console, *expiry* value is not null and indeed the document is deleted after 20 seconds. There is also a *touch*() method that allows to modify *expiry* without changing the document.

Another value that is stored on server is CAS (Compare And Swap). This value can help to control the behavior in case of concurrent document updates and is modified each time there is a change in the document. It can be passed as an optional parameter to the *insert*() method.

4.4.3 Updating documents

The *upsert*() method has exactly the same objective as *insert*() with the same mandatory and optional arguments. The difference is that, if the document with that ID already exists, the stored content will be overwritten, while using *insert*() would have raised an error. Similarly, a *replace*() method can replace an existing document by another and is present to efficiently handle concurrent modifications based on CAS value.

4.4.4 Retrieving documents

As we saw previously, we can get the information of a document by passing it's string ID to the *get*() method. However, there are other ways to do it.

```
JsonDocument libraryDoc = JsonDocument.create("doc_1", library); // library is a JsonObject like used before
bucket.upsert(libraryDoc); // inserting to bucket
System.out.println(bucket.get(libraryDoc).toString().substring(12)); // retrieving
```

In this case, ID is extracted from the provided document and we obtain a similar output.

```
System.out.println(bucket.get(libraryDoc, 30, TimeUnit.SECONDS).toString().substring(12));
```

We can also set a custom timeout before an error is thrown in case if it takes longer to retrieve a document than the specified time. This functionality is also available for creating, updating and deleting operations.

```
System.out.println(bucket.getAndLock(libraryDoc, 10).toString().substring(12));
bucket.upsert(libraryDoc); // raises concurrent.TimeoutException if executed before 10 seconds
```

In addition to getting a document, we can also lock it on write operation (reading is still possible) for a specified amount of time.

Finally, $getFromReplica(DocumentID)$ function allows to get replica copies returning an iterator to the requested documents. We can not test replication functionality since our virtual cluster consists of a single node.

4.4.5 Deleting documents

```
bucket.remove("doc_1")); // via ID
// or
bucket.remove(libraryDoc)); // via document
```

Nothing needs to be added here besides that these both methods will return the document's ID and the updated CAS value.

4.4.6 Operations on sub-documents

Sometimes, when only a small part in a document needs to be updated/added/retrieved, it is more efficient (in terms of network bandwidth) to work with this concerning part rather than with a whole document. A part of a document is called a *sub-document* and can be accessed via a specified path. For illustration, if we retake our library JsonDocument :

```
{
        "name": "BSH",
        "opening_hours": [
                "working_days 8h - 20h",
                "saturday 10h - 17h"
        ],
        "characteristics": {
                "seats": 2198,
                "floors": 8
        },
        "address": "Campus Solbosch"
}
```

Here, $name$, $opening\_hours[1]$ and $characteristics.seats$ are all correct paths. Below are shown some modifications that we can apply on sub-documents which by the way are atomic operations.

```
boolean createParents = true; // to create paths if they not exist
bucket.mutateIn("doc_1")
    .upsert("name", "BSH v.2", createParents)
    .upsert("street.number", 50, createParents)
    .arrayInsertAll("opening_hours[1]", "31 December 2017 holiday", "1 January 2018 holiday")
    .remove("characteristics.floors")
    .execute();

System.out.println(bucket.get("doc_1"));
```

The document now becomes :

```
{
      "name": "BSH v.2",
      "opening_hours": [
         "working_days 8h - 20h",
         "31 December 2017 holiday",
         "1 January 2018 holiday",
```

```
        "saturday 10h - 17h"
      ],
      "characteristics": {
        "seats": 2198
      },
      "address": "Campus Solbosch",
      "street": {
        "number": 50
      }
}
```

In addition to this, we can do reading and simple existence checking operations. Executed on the obtained document :

```
DocumentFragment<Lookup> res = bucket.lookupIn("doc_1")
                                              .get("name")
                                              .exists("characteristics.seats")
                                              .get("opening_hours[3]")
                                              .execute();
System.out.println(res);
```

Along with ID and CAS, the following information can be found in the output :

```
GET(name){value=BSH v.2}, EXIST(characteristics.seats){value=true},
GET(opening_hours[3]){value=saturday 10h - 17h}
```

4.4.7 Other useful methods

```
bucket.bucketManager().info(); // gets information of current bucket
bucket.bucketManager().flush(); // deletes all the contained data
bucket.bucketManager().createN1qlPrimaryIndex(true, false); // creates a primary index and is ignored if it
                                                            // already exists
```

4.4.8 N1QL syntax

Java SDK provides two ways for applications to execute N1QL queries. Either with domain-specific language (DSL) or by embedding the queries into a string.

In the following examples we used the *beer – sample* bucket that first must be activated from the Couchbase Web Console in the settings menu. It is composed of *beer* documents (some included fields are : "abv"[5], "brewery_id", "category", "description", "name", "style", "type") and *brewery* documents (containing "address", "city", "country", "geo", "description", "name", "type", "website", etc.)

```
// DSL
System.out.println(bucket.query(select("brewery_id").from(i("beer−sample")).where(x("name")
                                        .eq(s("Jupiler")).and(x("type").eq(s("beer"))))));

// same query in N1QL syntax
System.out.println(bucket.query(N1qlQuery.simple("SELECT brewery_id FROM ‘beer−sample‘ "
                                        + "WHERE name = ’Jupiler’ AND type = ’beer’"")));
```

Output is the same in both cases except for the execution times that are variable and do not allow to conclude which approach is faster.

[5] Alcohol by volume.

```
N1qlQueryResult{status='success', finalSuccess=true, parseSuccess=true, allRows=[{
          "brewery_id": "inbev"
       }], signature={"brewery_id":"json"}, info=N1qlMetrics{resultCount=1,
      errorCount=0, warningCount=0, mutationCount=0, sortCount=0, resultSize=45,
      elapsedTime='801.210398ms', executionTime='801.138715ms'}, profileInfo={},
      errors=[], requestId='9d8a84f2-0301-49e3-ac39-7c0a1b42d40b', clientContextId=''}
```

DSL has the advantage of for example preventing to send erroneous queries since the syntax can be checked at the compilation time. However, it might seem less intuitive for novice users, that's why here we will focus on the other solution which is more SQL-like.

Selecting first 2 beer documents that contain "Belgian" in their description and have more than 7.5% of alcohol.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT name, abv, description " +
                                                 "FROM `beer-sample` " +
                                                 "WHERE description LIKE '%Belgian%' " +
                                                 "AND type = 'beer' AND abv > 7.5 " +
                                                 "LIMIT 2")));
```

Result :

```
{
       "abv": 8,
       "description": "Our first anniversary release is a Belgian-style strong ale
           that is amber in color, with a light to medium body. Subtle malt sweetness
           is balanced with noticeable hop flavor, light raisin and mildly spicy, cake
           -like flavors, and is finished with local wildflower honey aromas. Made
           with 80% Organic Malted Barley, Belgian Specialty grains, Forbidden Fruit
           yeast, domestic hops and Round Rock local wildflower honey, this beer is
           deceptively high in alcohol. ",
       "name": "One"
       }, {
       "abv": 10.5,
       "description": "Our silver medal winning Belgian style Holiday beer brewed with
            dark Belgian candied sugar and special spices. This brew has a spicy aroma
            and flavor with a sweet malt taste. At 10.5% alcohol it is sure to warm
           you up during the Holiday season.",
       "name": "Rude Elf's Reserve"
}
```

Selecting brewery_id from documents that do contain that field, in descending order.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT DISTINCT brewery_id FROM `beer-sample` "
                                                 + "WHERE brewery_id IS NOT MISSING "
                                                 + "ORDER BY brewery_id DESC LIMIT 3")));
```

Result :

```
{
       "brewery_id": "zea_rotisserie_and_brewery"
       }, {
       "brewery_id": "yuksom_breweries"
       }, {
       "brewery_id": "yuengling_son_brewing"
}
```

Join beer documents (b1) with their corresponding breweries (b2) where the primary key to b2 is in the brewery_id field of b1.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT b1.name AS beer_name, "
                                                 + "b2.name AS brewery_name, b2.website "
                                                 + "FROM `beer-sample` AS b1 "
                                                 + "JOIN `beer-sample` AS b2 ON KEYS b1.brewery_id "
                                                 + "WHERE b1.type = 'beer' LIMIT 2")));
```

Result :

```
    {
        "beer_name": "21A IPA",
        "brewery_name": "21st Amendment Brewery Cafe",
        "website": "http://www.21st-amendment.com/"
    }, {
        "beer_name": "563 Stout",
        "brewery_name": "21st Amendment Brewery Cafe",
        "website": "http://www.21st-amendment.com/"
    }
```

Listing breweries per beer name and skipping the first five results.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT beer.name, "
                                                 + "ARRAY_AGG(brewery.name) brewery_list "
                                                 + "FROM `beer-sample` beer "
                                                 + "INNER JOIN "
                                                 + "`beer-sample` brewery "
                                                 + "ON KEYS beer.brewery_id "
                                                 + "GROUP BY beer.name LIMIT 2 OFFSET 5")));
```

Result :

```
{
    "brewery_list": [
           "Kiuchi Shuzou Goushi Kaisya"
    ],
    "name": "Hitachino Nest Japanese Classic Ale"
    }, {
    "brewery_list": [
           "Aksarben Brewing (BOP)"
    ],
    "name": "Bourbon Imperial Stout"
}
```

Listing beers per breweries from Belgium.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT brewery.name, "
                                                 + "ARRAY_AGG(beer.name) beer_list "
                                                 + "FROM `beer-sample` beer "
                                                 + "INNER JOIN "
                                                 + "`beer-sample` brewery "
                                                 + "ON KEYS beer.brewery_id "
                                                 + "WHERE brewery.country = 'Belgium' "
                                                 + "GROUP BY brewery.name LIMIT 2")));
```

Result :

```
    {
        "beer_list": [
           "Trappist Blond",
           "Trappist Bruin Bier / Biere Brune",
           "Trappist Extra"
```

```
        ],
        "name": "Brouwerij De Achelse Kluis"
    }, {
        "beer_list": [
            "Framboise 1997",
            "Geuze Boon",
            "Kriek",
            "Mariage Parfait 1995",
            "Pertotale Faro"
        ],
        "name": "Brouwerij Boon"
    }
```

Number of distinct beers produced by breweries around Brussels.[6]

```
System.out.println(bucket.query(N1qlQuery
        .parameterized("SELECT count(DISTINCT beer.name) AS total_beers "
                        + "FROM `beer-sample` beer "
                        + "INNER JOIN "
                        + "`beer-sample` brewery "
                        + "ON KEYS beer.brewery_id "
                        + "WHERE brewery.name WITHIN "
                                + "(SELECT name "
                                + "FROM `beer-sample` WHERE type = 'brewery' AND "
                                + "ABS(geo.lat - $1) <= 0.2 AND "
                                + "ABS(geo.lon - $2) <= 0.2)",
                        JsonArray.from(50.8503, 4.3517)))); // Brussels coordinates
```

Result :

```
    {
        "total_beers": 65
    }
```

Average time to run this queries, without the "LIMIT" parameter, is around 1.4 sec which is rather impressive considering the dataset size (7303 documents) and our not so fast machine (Intel i3 CPU @ 1.80GHz with 4Gb of RAM). There are more intelligent ways to parse results returned from queries as well as a possibility to set parameters to queries (for example consistency level) but we do not cover it in this report.

To try to achieve even lower execution times of queries, like was said previously, one way of doing it is by creating GSI. For example lets consider the following query which in best case takes around 650ms to execute.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT name "
                                                 + "FROM `beer-sample` "
                                                 + "WHERE type = 'brewery' AND website LIKE '%.be/'"))
                                                 .info().executionTime());
```

It simply finds breweries names where the website ends by ".be/" but here we only print the execution time. To create a GSI to index for example type and website attributes it can be done as follows :

[6] One interesting question during our presentation was "What are the units?". In fact, the "geo" argument has "lat"/"lon" fields but also has an "accuracy". The latter simply indicates the precision of the coordinates and has values like "ROOFTOP", "RANGE_INTERPOLATED" or "APPROXIMATE". Maybe this does not answer directly the question but probably our query should use this information to make the results more consistent with each other.

```
bucket.query(N1qlQuery.simple("CREATE INDEX type_website_index "
                              + "ON `beer-sample`(type, website) USING GSI"));
```

Now, if we re-run the previous query, the execution time becomes around 150ms. This way, we can greatly improve the rapidity of all the previous queries by creating appropriate GSI which can take some seconds to run but this only needs to be done once. It is possible to combine the previous command with a WHERE clause (to limit the number of documents being indexed) and with a WITH clause (to set the number of replicas or explicitly specify the nodes where they should be kept on). We haven't said it before but replication mechanism is also available for indexes.

The queries also allow to manage the database through traditional INSERT, UPDATE, DELETE statements.

Inserting two new documents. The first has an id = 'key1', fields ={'library', 'campus'} and the second has id = 'key2' and a field 'university'.

```
System.out.println(bucket.query(N1qlQuery.simple("INSERT INTO `beer-sample` (KEY, VALUE) " +
                                                 "VALUES ('key1', " +
                                                 " {'library': 'BSH', " +
                                                 " 'campus': 'Solbosch' })," +
                                                 "VALUES ('key2', { 'university': 'ULB'})")));
```

The INSERT query can also be combined with the SELECT clause allowing to insert documents formed from existing fields. The fields can also be searched from an external bucket. In that case, UUID() parameter for KEY argument will generate unique keys for each new created document.

If one day Belgium becomes a supreme nation that rules the world, we can then execute the following query where all the foreign breweries will be marked as belonging to our territory. Additionally, we return the name field of all the updated documents.

```
System.out.println(bucket.query(N1qlQuery.simple("UPDATE `beer-sample` "
                                                 + "SET country = 'Belgium' "
                                                 + "WHERE type = 'brewery' AND country <> 'Belgium' "
                                                 + "RETURNING name")));
```

There is also an UPSERT query that has an INSERT-like syntax, giving a key-value pair. It is a combination of UPDATE and INSERT. If a document referenced by the provided key already exists in the database, it is updated with the given value. Otherwise, a new document is created.

Deleting a document by its id.

```
System.out.println(bucket.query(N1qlQuery.simple("DELETE FROM `beer-sample` "
                                                 + "USE KEYS '21st_amendment_brewery_cafe'")));
```

Deleting the first ten documents corresponding to a certain criteria and returning all the deleted content.

```
System.out.println(bucket.query(N1qlQuery.simple("DELETE FROM `beer-sample` "
                                                 + "WHERE type = 'beer' and abv = 0 "
                                                 + "LIMIT 10 "
```

```
                                                    + "RETURNING *")));
```

4.4.9 N1QL performance

To study (one aspect of) the system performance, the execution time of an N1QL query is calculated for a varying number of documents.

The chosen query selects the *name*, *category* and *brewery_id* fields of *beer* having their alcohol by volume above 5.

```
System.out.println(bucket.query(N1qlQuery.simple("SELECT name, category, brewery_id "
                                                 + "FROM `beers-sample` "
                                                 + "WHERE type = 'beer' AND abv > 5.0"))
                                                 .info().executionTime());
```

It is executed on the same *beers* – *sample* dataset as before, which contains approximately 5000 documents. This dataset is multiplied artificially, simply by duplicating documents and only changing their id. The content of the documents remains unchanged but it does not matter for the purpose of the task. At each duplication step, the size of the dataset doubles and the query execution time is calculated. The data are loaded into the database using the *cbdocloader* tool in command line, specifying from which directory they should be taken, and to which bucket they should be sent.

```
cbdocloader -n [host]:8091 -u [Admin] -p [pwd] -b [bucket-name] [directory]
```

Let us note that the biggest handled dataset is limited to 200000 documents, as the loading duration was already quite long (around 30 minutes for 200000 documents). This time could probably be reduced by tuning the loading tool or the settings of the database, for extensive studies to be led.

The results are shown in the following graph.

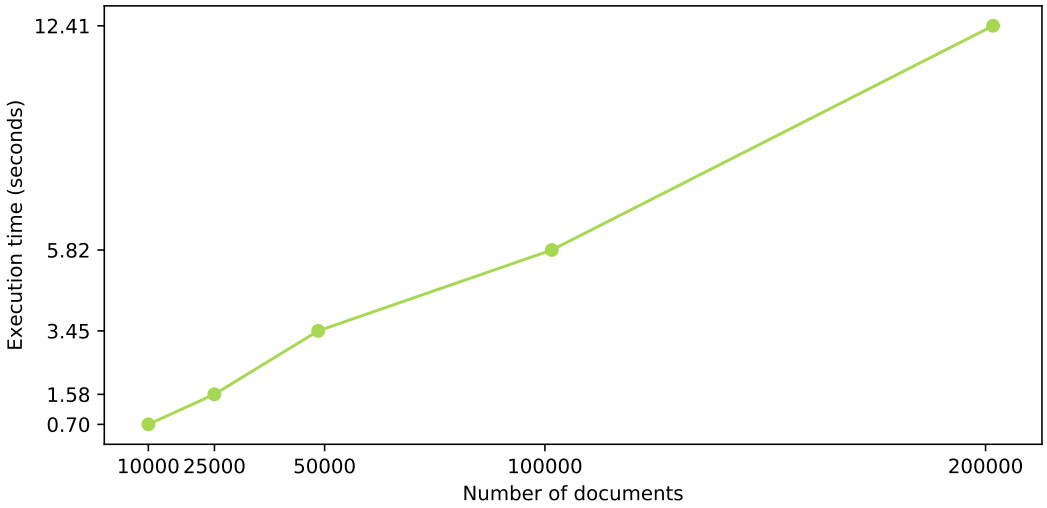

Figure 4.5: Query execution time according to the number of documents

The shape of the graphe is linear, which is not so bad, but it means that each time the data doubles, the time to execute the query approximately doubles too. This result is to put into

consideration that only a primary index was created at the time of the query execution. We expect that creating a global secondary index beforehand would give better performance, and the size of the data should not matter that much in this setting.

4.5 Conclusion

To conclude, this last chapter has outlined the core functionalities of Couchbase. We described its general architecture along with some main properties. An emphasis was done on the available possibilities from the programmer/administrator side. We saw that from this point of view, Chouchbase is rather flexible (has often many solutions to do the same thing) and offers plenty options to work with data along with the management and supervision of cluster and nodes.

Moreover, there are many other functionalities that were not discussed or shown in this report, such as custom MapReduce functions, manipulation of non JSON documents (i.e. key-value storage), batching operations (where the requests and responses are done via a pipelined communication which increases network throughput), full text search, users and clusters management, security, replication and others.

In sum, Couchbase seems to be a good Document Store candidate for small to big businesses providing an efficient and reliable system. It may be especially suited for societies willing to work with JSON format, willing or planning to expand their database to mobile/IoT applications and where the developers want to get used to the tool quickly and prefer to have an SQL like query language. It is hard to tell whether Couchbase is better than the solutions discussed in section 3.2. All of them share similar characteristics and seem to provide more or less similar services. To really see which solution is best for a given company, we think, it is necessary to deploy those systems and use it for some time in a practical environment.

Bibliography

[1] Guy Harrison, *"Next Generation Databases"*, Springer Science+Business Media New York, 2015.

[2] ACID definition, "http://searchsqlserver.techtarget.com/definition/ACID", consulted on 4/10/2017.

[3] Dan McCready and Ann Kelly, *"Making Sense of NoSQL"*, Manning Publications, 2014.

[4] Big Data evolution graph, "http://www.atkearney.be/analytics/ideas-insights/article/-/asset_publisher/hZFiG2E3WrIP/content/big-data-and-the-creative-destruction-of-today-s-business-models/10192", consulted on 4/10/2017.

[5] Graph Databases for Beginners: ACID vs. BASE Explained, "https://neo4j.com/blog/acid-vs-base-consistency-models-explained/", consulted on 5/10/2017.

[6] Abandoning ACID in Favor of BASE, "https://www.thoughtco.com/abandoning-acid-in-favor-of-base-1019674", consulted on 5/10/2017.

[7] What is a Key-Value Store?, "http://www.aerospike.com/what-is-a-key-value-store/", consulted on 7/10/2017.

[8] NoSQL Databases Explained, "https://www.mongodb.com/nosql-explained", consulted on 9/10/2017.

[9] What is a Document Store Database?, "http://database.guide/what-is-a-document-store-database/", consulted on 9/10/2017.

[10] Document-oriented database, "https://en.wikipedia.org/wiki/Document-oriented_database", consulted on 9/10/2017.

[11] Document Databases Explained, "http://basho.com/resources/document-databases/", consulted on 9/10/2017.

[12] Comparing document-oriented and relational data, "https://developer.couchbase.com/documentation/server/3.x/developer/dev-guide-3.0/compare-docs-vs-relational.html", consulted on 10/10/2017.

[13] JSON: The Fat-Free Alternative to XML, "http://www.json.org/xml.html", consulted on 10/10/2017.

[14] Entity Relationships and Document Design, "https://developer.couchbase.com/documentation/server/4.6/data-modeling/entity-relationship-doc-design.html", consulted on 10/10/2017.

[15] DB-Engines Ranking of Document Stores, "https://db-engines.com/en/ranking/document+store", consulted on 11/10/2017.

[16] MongoDB Architecture, "https://www.mongodb.com/en/mongodb-architecture", consulted on 11/10/2017.

[17] Introduction to MongoDB, "https://docs.mongodb.com/manual/introduction/", consulted on 11/10/2017.

[18] Amazon DynamoDB, "https://aws.amazon.com/dynamodb/", consulted on 11/10/2017.

[19] Apache CouchDB 2.1 Documentation, "http://docs.couchdb.org/en/latest/", consulted on 11/10/2017.

[20] Regions Azure, "https://azure.microsoft.com/fr-fr/regions/", consulted on 11/10/2017.

[21] Welcome to Azure Cosmos DB, "https://docs.microsoft.com/en-us/azure/cosmos-db/introduction", consulted on 11/10/2017.

[22] MarkLogic key features, "http://www.marklogic.com/what-is-marklogic/features/", consulted on 11/10/2017.

[23] Couchbase product editions, "https://www.couchbase.com/products/editions", consulted on 14/10/2017.

[24] Couchbase Mobile, "https://www.couchbase.com/products/mobile", consulted on 14/10/2017.

[25] Couchbase Server What's New?, "https://developer.couchbase.com/documentation/server/5.0/introduction/whats-new.html", consulted on 14/10/2017.

[26] Couchbase Server, Distributed Data Management, "https://developer.couchbase.com/documentation/server/5.0/concepts/distributed-data-management.html", consulted on 15/10/2017.

[27] REST API endpoint list, "https://developer.couchbase.com/documentation/server/5.0/rest-api/rest-endpoints-all.html", consulted on 14/10/2017.

[28] Couchbase Server: An Architectural Overview, "https://info.couchbase.com/Architectural-Overview_Guided-LP.html", consulted on 25/10/2017.