Final thesis

Exploration of NoSQL Technologies for managing hotel reservations

by

Sylvain Coulombel

LiTH-IDA/ERASMUS-A--15/001--SE

2014-11-22
Final Thesis

Exploration of NoSQL Technologies for managing hotel reservations

by

Sylvain Coulombel

LiTH-IDA/ERASMUS-A--15/001--SE

2014-11-22

Supervisor: Emmanuel Varoquaux, Amadeus IT Group
Examiner: Patrick Lambrix, Linköping University
Exploration of NoSQL technologies for managing hotel reservations

Author:
Sylvain Coulombel

University Supervisor:
Patrick Lambrix

Company:
Amadeus IT Group

Company Supervisors:
Emmanuel Varoquaux
Mario Corchero
Pedro Arnedo

A thesis submitted in the framework of a double degree between Compiègne University Of Technology (UTC) and Linköping University (LiU)

November 2014
During this project NoSQL technologies for Hotel IT have been evaluated. It has been determined that among NoSQL technologies, document database fits the best this use-case. Couchbase and MongoDB, the two main documents stores have been evaluated, their similarities and differences have been highlighted. This reveals that document-oriented features were more developed in MongoDB than Couchbase, this has a direct impact on search of reservations functionality. However Couchbase offers a better way to replicate data across two remote data centers. As one of the goals was to provide a powerful search functionality, it has been decided to use MongoDB as a database for this project. A proof of concept has been developed, it enables to search reservations by property code, guest name, check-in date and check-out date using a REST/JSON interface and confirms that MongoDB could work for storing hotel reservations in terms of functionality. Then different experiments have been conducted on this system such as throughput and response time using specific hotel reservation search query and data set. The results we got reached our targets. We also performed a scalability test, using MongoDB sharding functionalities to distribute data across several machines (shards) using different strategies (shard keys) so as to provide configuration recommendations. Our main finding was that it was not necessary to always distribute the database. Then if "sharding" is needed, distributing the data according to the property code will make the database go faster, because queries will be sent directly to the good machine(s) in the cluster and thus avoid "scatter-gather" query. Finally some search optimizations have been proposed, and in particular how an advanced search by names could be implemented with MongoDB.
Acknowledgements

Writing a master thesis is not an easy task and this would not have been possible without all the help I receive.

Firstly I would like to thank Emmanuel, Catalin, François, Kevin, Pedro, Pierre, Mario from the reservation team for their warm welcome, and the interest they had in the project.

In particular I would like to express my gratitude to my mentors Mario Corchero and Perdro Arnedo. Thank you for your comments, advice, and availability during the whole project, I learnt a lot from you. I would like to thank my supervisor Emmanuel Varoquaux for his continuous feedback, support and helpful guidance he gave me during this work. I would also like to say a BIG thank you to Pierre Lombard and Melinda Mindruta for sharing with me their technical expertise, ideas and giving me recommendations which helped me a lot.

I am also very grateful to Attila Tozser and Manuel Swiercz from Amadeus Germany for helping me to deploy MongoDB. I would also like to thank Sebastien Charpentier for providing me the opportunity to work in his department.

I would like to thank Patrick Lambrix, my supervisor at Linköping University for supervising this work and whose class made me want to learn more on databases field.

Lastly I would like to thank my family. Thank you for your love and support throughout my studies.
2.4 Current solution for storing hotel reservations ........................................... 15
  2.4.1 e-voucher image ..................................................................................... 15
  2.4.2 Current storing solution .......................................................................... 16
2.5 Choice of a kind of NoSQL database ............................................................. 16

3 Document store ............................................................................................... 17
  3.1 Couchbase and MongoDB overview ............................................................ 17
    3.1.1 History .................................................................................................... 17
    3.1.2 Kind of database and the value they store ............................................. 18
      3.1.2.1 MongoDB: A full document store ....................................................... 18
      3.1.2.2 Couchbase: From a key-value store to a document store ............... 19
      3.1.2.3 Both are schemaless ........................................................................... 19
  3.2 Couchbase and MongoDB functionality study ................................................. 20
    3.2.1 Database connection .............................................................................. 20
      3.2.1.1 Couchbase ......................................................................................... 20
      3.2.1.2 MongoDB ......................................................................................... 20
    3.2.2 Document insertion ................................................................................ 20
      3.2.2.1 Couchbase set() method ................................................................. 20
      3.2.2.2 MongoDB insert method ................................................................. 21
    3.2.3 Document retrieve .................................................................................. 21
      3.2.3.1 Couchbase get() method ................................................................. 21
      3.2.3.2 MongoDB find_one() method ......................................................... 21
      3.2.3.3 A functional retrieve (on a different field from key or id) ............... 21
    3.2.4 Document search .................................................................................... 22
      3.2.4.1 MongoDB find() method ................................................................. 22
      3.2.4.2 Couchbase views .............................................................................. 23
      3.2.4.3 Comparison ....................................................................................... 24
    3.2.5 Document aggregation ........................................................................... 24
      3.2.5.1 MongoDB aggregate method ............................................................. 25
      3.2.5.2 Couchbase views .............................................................................. 25
      3.2.5.3 MongoDB Aggregation framework limitation .................................... 26
    3.2.6 Special indexes ...................................................................................... 27
  3.3 Couchbase and MongoDB Architecture ....................................................... 27
    3.3.1 MongoDB architecture ......................................................................... 28
      3.3.1.1 Replication ......................................................................................... 28
        Read and write ........................................................................................... 29
        Failover ...................................................................................................... 29
        Write concern ............................................................................................ 30
        Journaling .................................................................................................. 31
        Read isolation ............................................................................................ 33
      3.3.1.2 Sharding ............................................................................................. 33
        Architecture ................................................................................................ 33
        Query routing .............................................................................................. 33
        Kind of shard key ...................................................................................... 34
        Data splitting ............................................................................................. 35
        Query isolation ........................................................................................... 35
      3.3.1.3 Process and physical machine ........................................................... 36
3.3.2 Couchbase architecture  ........................................ 37
  3.3.2.1 Sharding ........................................ 37
     Architecture ........................................ 37
     Data splitting ....................................... 37
     Query routing ....................................... 38
     Kind of shard key .................................. 39
     Query isolation ..................................... 39
  3.3.2.2 Process and physical machine ....................... 39
  3.3.2.3 Replication ..................................... 39
     Read and write ...................................... 39
     Failover ............................................ 40
     Write concern ....................................... 40
     Read isolation ....................................... 40
  3.3.2.4 XDCR replication .................................. 41
3.3.3 How we can we bring data closest to customers without XDCR using MongoDB?  ........................................ 43
  3.3.3.1 Use replica set in two different data-centers  .......... 43
  3.3.3.2 Use tag aware sharding ............................ 43
  3.3.3.3 Combine the two .................................. 44
3.4 MongoDB and Couchbase mapping .......................... 45
3.5 Conclusion .................................................. 46
  3.5.1 Functionality ........................................ 46
  3.5.2 Architecture ......................................... 46
     3.5.2.1 CAP Theorem and BASE .......................... 46
     3.5.2.2 Synthesis ..................................... 47
     3.5.2.3 Recommendations .............................. 47
4 MongoDB prototype .......................................... 51
  4.1 Data model ............................................... 51
     4.1.1 NoSQL data modelling ............................. 51
     4.1.2 Our modeling ..................................... 53
  4.2 Choice of a Framework and API ........................... 54
  4.3 Supported operations by the prototype ................. 54
     4.3.1 Search by ID : /searchByID ....................... 55
     4.3.2 Search by check in date and check out date:/searchByRangeDate 55
     4.3.3 Search by name: /searchByName .................... 56
     4.3.4 Search by transaction date ....................... 57
     4.3.5 Search by multi: /searchByMulti .................. 57
  4.4 Implementation .......................................... 57
  4.5 Creating good indexes .................................... 58
     4.5.1 Theory .......................................... 58
     4.5.2 Strategy ......................................... 59
  4.6 Conclusion ............................................... 60
5 Performance tests .......................................... 61
  5.1 Testing framework and configuration used ............ 61
     5.1.1 Testing framework ............................... 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1.1</td>
<td>Populate the database</td>
<td>61</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>Query generator</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Methodology</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Workload in Search definition</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Range date distribution</td>
<td>63</td>
</tr>
<tr>
<td>5.1.1.3</td>
<td>Indexing</td>
<td>63</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Search and Insert</td>
<td>64</td>
</tr>
<tr>
<td>5.1.3</td>
<td>The different Sharding configurations</td>
<td>64</td>
</tr>
<tr>
<td>5.1.3.1</td>
<td>Increase shard number</td>
<td>64</td>
</tr>
<tr>
<td>5.1.3.2</td>
<td>Different shard keys strategies</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Shard on hash of the id</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Shard on hash of the property code</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Justification</td>
<td>65</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Client and MongoS configuration</td>
<td>66</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Hardware and sofware configuration</td>
<td>66</td>
</tr>
<tr>
<td>5.1.5.1</td>
<td>Hardware configuration</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Machine</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Network</td>
<td>67</td>
</tr>
<tr>
<td>5.1.5.2</td>
<td>Software configuration</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Sharding not activated: Throughput and response time test</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Insertion time</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>Hypothesis</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>Results</td>
<td>68</td>
</tr>
<tr>
<td>5.2.1.3</td>
<td>Interpretation</td>
<td>69</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Search performances</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2.1</td>
<td>Hypothesis</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2.2</td>
<td>Results</td>
<td>71</td>
</tr>
<tr>
<td>5.2.2.3</td>
<td>Interpretation</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Scalability tests</td>
<td>71</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Insertion</td>
<td>71</td>
</tr>
<tr>
<td>5.3.1.1</td>
<td>Hypothesis</td>
<td>71</td>
</tr>
<tr>
<td>5.3.1.2</td>
<td>Results</td>
<td>72</td>
</tr>
<tr>
<td>5.3.1.3</td>
<td>Interpretation</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Search</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2.1</td>
<td>Hypothesis</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2.2</td>
<td>Results</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Shard key was hash-id</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>Shard key was property code</td>
<td>74</td>
</tr>
<tr>
<td>5.3.2.3</td>
<td>Interpretation</td>
<td>74</td>
</tr>
<tr>
<td>5.3.2.4</td>
<td>Comments about RAM size</td>
<td>75</td>
</tr>
<tr>
<td>5.4</td>
<td>Maximum throughput test in a sharded environment</td>
<td>75</td>
</tr>
<tr>
<td>5.5</td>
<td>Criticism of test protocol and modification</td>
<td>76</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Problem identified</td>
<td>76</td>
</tr>
<tr>
<td>5.5.1.1</td>
<td>Workload and measure are dependent</td>
<td>76</td>
</tr>
<tr>
<td>5.5.1.2</td>
<td>Throughput and response time are dependent on time</td>
<td>77</td>
</tr>
<tr>
<td>5.5.1.3</td>
<td>Use of workload generator and measuring process on same machine</td>
<td>77</td>
</tr>
</tbody>
</table>
6.2.2.3 Benchmarking the three methods ........................................ 97
6.2.2.4 Sort results by relevance ............................................. 97
6.2.2.5 Benchmark with a sort constraints ................................... 98
6.2.2.6 Going further .............................................................. 98

Store all names related to a booking in a text or array index 98
Phonetic search ................................................................. 98
Weight on text indexes ......................................................... 99

6.3 Conclusion ................................................................. 99

7 Chapter conclusion .......................................................... 101
7.1 Conclusion ................................................................. 101
7.2 Future work ................................................................. 102

A A booking example .......................................................... 103

B Different kinds of indexes existing in MongoDBs ...................... 107

C Complex sort and compound indexes .................................... 109
C.1 Sort is done on fields that are not in the query ...................... 109
C.2 Sort on two different fields .............................................. 110

D Search by name(s) optimizations implementation .................... 111
D.1 Text index and sort by score ............................................. 111
D.2 Array index and sort by score using the aggregation framework . 111

E Search by 2 ranges of date implementation ............................. 113
E.1 Room stay dates in the document ....................................... 113
E.1.1 Compound index ....................................................... 113
E.1.2 Geo2D index ............................................................. 113
E.2 Index creation ............................................................... 114
E.2.1 Compound index ....................................................... 114
E.2.2 Geo2D index ............................................................. 114
E.3 Data querying ............................................................... 114
E.3.1 Compound index ....................................................... 114
E.3.2 Geo2D index ............................................................. 115

Bibliography ................................................................. 117
List of Figures

1.1 Booking process at AHP ........................................... 4
2.1 Vertical and Horizontal scalability ............................... 8
2.2 NoSQL database and CAP theorem ............................. 11
2.3 Partitioning and replication of keys in Dynamo ring ........... 13
2.4 The 4 different categories of NoSQL databases ............... 16
3.1 MongoDB data model .............................................. 18
3.2 MongoDB: a replica set ........................................... 28
3.3 MongoDB: Journaling in MongoDB ............................. 32
3.4 MongoDB: a MongoDB sharded cluster ....................... 33
3.5 Couchbase architecture ........................................... 37
3.6 Couchbase: Mapping of documents key to partitions and then to server 38
3.7 Couchbase architecture ........................................... 40
3.8 Couchbase Cross Data Center Replication (XDCR) ............ 41
3.9 Replicas set distributed in two data-centers ................... 43
3.10 Combine distributed sharding and replicas ................. 44
3.11 If master-master replication was possible in MongoDB ... 46
3.12 Couchbase and MongoDB comparison ....................... 48
5.1 Configuration summary ........................................... 67
5.2 Insertion time with index and no index when sharding is not enabled 69
5.3 Insertion throughput with index and no index when sharding is not enabled 69
5.4 Search response time ............................................ 70
5.5 Search throughput ................................................. 70
5.6 Insertion response time with index with sharding disabled and when the cluster was sharded on hash-id with the three and six shards configuration ........................................ 72
5.7 Insertion throughput with sharding disabled and when the cluster was sharded on hash-id with the three and six shards configuration ... 72
5.8 Search response time with sharding disabled and when the cluster was sharded on hash-id with the three and six shards configuration ... 73
5.9 Search response time with sharding disabled and when the cluster was sharded on the property code with the three and six shards configuration 74
5.10 Configuration summary ........................................... 78
5.11 Shard on hash-id and property code comparison, response time distribution according to throughput .............................. 79
List of Figures

5.12 Shard on property code, response time distribution according to increasing throughput ........................................ 81
5.13 Shard on property code, response time distribution according to increasing throughput with two different workloads .............................. 82
5.14 Shard on property code, response time distribution according to increasing throughput with sending queries to one and two Mongoose .......................................................... 83

6.1 Search in a range of two dates using geospatial indexes .................. 94
6.2 Comparison of the three strategies response time .......................... 97
6.3 Comparison of the three strategies response time (zoom) ................. 97
6.4 Comparison of text indexes and array indexes (aggregation) response  
time when sorting results by relevance ........................................... 98
List of Tables

3.1 Data is split into chunks, chunks are distributed in different shards . . 35
3.2 Three replica sets on three machines . . . . . . . . . . . . . . . . . . 36

4.1 A customers table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.2 A bookings table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.3 An hotels table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

5.1 Physical machine configuration . . . . . . . . . . . . . . . . . . . . . . . 67
5.2 Network latency between the different machines . . . . . . . . . . . . . 67
5.3 Response time comparison table between hash-id and property code
shard keys strategy under an increasing workload . . . . . . . . . . . . . 80

6.1 Index order strategy, Property is the first key in index 1 and last key in
index 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
6.2 The three indexes strategy . . . . . . . . . . . . . . . . . . . . . . . . . 96
Abbreviations

ACID  Atomicity, Consistency, Isolation, Durability
AHP   Amadeus Hotel Platform
API   Application Programming Interface
BASE  Basically Available, Soft state, Eventual consistent
BSON  Binary JSON
CAP   Consistency, Availability, Partition
CPU   Central Process Unit
CRS   Computer/Central Reservation System
CRUD  Create, Read, Update, Delete
CX    Cancellation number
DBA   Database Administrator
DBMS  Database Management System
DB    Database
GDS   Global Distribution System
JSON  JavaScript Object Notation
PNR   Passenger Number Record
POC   Proof Of Concept
RAM   Random Access Memory
REST  REpresentational State Transfert
RDBMS Relational Database Management System
SLA   Service-Level Agreement
SSD   Solid-state Drive
SQL   Structured Query Language
TPS   Transactions Per Second
XDCR  Cross Data Center Replication
Chapter 1

Introduction

1.1 Context

This master thesis project was hosted at Amadeus Hotel Platform\(^1\) in the Research and Development department in Sophia-Antipolis (France) site.

1.1.1 The company

Amadeus [1] is an IT company dedicated to the travel industry, present in 195 countries with a worldwide team of about 11,000 people. The goal of their IT solution is to help improve the business performance of travel agencies, companies, airlines, airports, hotels, railways. The company was founded in 1987 as a neutral global distribution system (GDS) by Air France, Iberia, Lufthansa and SAS, the aim was to establish the link between provider’s content (Airlines), travel agencies and consumers. Amadeus diversifies in new businesses (New Business Unit) which are hotel (Amadeus Hotel), rail (Amadeus Rail) and airport (Airport IT).

1.1.2 Amadeus Hotel

Amadeus Hotel acts as an intermediary between Hotel and Customer (Travel agency, booking web site). It provides distribution services to a large network of travel agencies and online channels, performing around 40,000 bookings per day with a content of more than 700,000 hotel properties worldwide. The Hotel Platform also provides

\(^1\)Part of the New Business Unit.
a complete portfolio of IT solutions for hotel chains, including a Central Reservation System (CRS), a Property Management System, a Call Center application, an Internet Booking Engine, and a Business Intelligence module.

1.2 Thesis work

This section introduces the aim of the project.

1.2.1 Project motivation

The Hotel Platform is in constant evolution to answer to the complexity of the distribution market and to fulfill the needs of very demanding IT customers such as high service-level agreement (SLA). Facing these challenges, Amadeus actively explores new solutions for storing and managing reservation data, as a complement to traditional relational databases, to conciliate data redundancy, the best possible performances and the cheapest cost per stored gigabyte.

1.2.2 Project goal

Relational databases are a very powerful tool and have been very popular since 1970’s. Unlike software, hardware and even programming language, it seems that relational databases have always lived in a stable environment and are still not dead. In the past a software architect had to decide which relational database to use, but did not ask so far which kind of database to use.

However, today a new kind of databases called NoSQL databases, is emerging. These databases are designed to be horizontally scalable by running on multiple machines. They also tend to solve the impedance mismatch problem. Unlike traditional RDBMS\(^2\), NoSQL databases break the ACID rules (will be defined in 2.2.2), which was one of the most important and oldest concepts of database theory. Major players of the Internet industry have already adopted, or are behind this new kind of database such as Facebook (HBase), Amazon (DynamoDB), Twitter (Cassandra) and Google (BigTable). Today Amadeus manages hotel booking by storing data in a relational database (using Oracle) and is now considering using NoSQL technology.

The goal of this project is to study what NoSQL technologies can bring for managing hotel reservations, and try to answer this question for a chosen NoSQL solution:

\(^2\)Relational Database Management System
• Data consistency: What does NoSQL guarantee in terms of data consistency?

• Crisis management: What happens in case of node failure or entire cluster failure?

• Response time: How long will it take for the system to return results?

• Scaling: How will the system react to a huge amount of requests or a huge amount of data?

• Scaling out: NoSQL databases are known to be horizontally scalable. Can it be an advantage for Hotel business?

• Query: Can we express complex queries and in an easy way?

1.2.3 Booking process at Amadeus

To better understand the use case, here is the description of the booking process (see figure 1.1). When a consumer or a travel agency is booking a hotel, several steps are needed.

• A UI access should be available. This can be a web page (for instance Expedia.com or the Amadeus Selling Platform) where the user can enter information such as the date and the location where he would like to book a room.

• Afterwards the system queries the availability database and displays a list of all available hotels.

• Then the user would like some more information about an available room (picture, description, translation in other language). This information is contained in the content database. The price is also determined by the pricing module which computes the real prices according to rules given by the providers (period of the year, number of available room ...).

• Finally when the user has made his decision, the booking is created (sell transaction) and stored in the booking database.

Storing the booking in the database is at the heart of the hotel reservation process. This booking acts as a contract between the customers and the providers. The customer should also be able to modify a booking (modify transaction) and cancel an existing booking (cancel transaction). When the booking is recorded in the database, the travel agency or customer should be able to retrieve the booking by
simple ID such as confirmation number, cancellation number or PNR (passenger number record) record locator\(^3\). However we should also be able to realize complex search. For instance a hotel or the call center should be able to find all bookings of the customers coming in the hotel between two given dates, or find a customer by its name(s). The customer should be able to retrieve his booking. The thesis work will concentrate on that booking database.

![Figure 1.1: Booking process at AHP](image)

**1.2.4 Thesis work approach**

To find an answer to questions in 1.2.2 we will start by gathering general information about NoSQL technology, compare existing open source solutions using literature and test of systems and select the most appropriate for the prototype. Then a prototype will be developed using this technology and integrating pieces of the current Hotel reservation search and retrieve software to provide a simplified but realistic environment. Finally, a performance benchmark will be done to validate the architecture and provide recommendations for the future implementations based on this proof of concept.

**1.2.5 Thesis Outline**

The thesis is organized as follows:

- Chapter 2 presents the NoSQL world and theoretical background.
- Chapter 3 compares the two main NoSQL document stores: MongoDB and Couchbase.

---

\(^3\)Pointer to the PNR, which contains data about a specific reservation such as flight, hotel and train.
• Chapter 4 describes the MongoDB search of reservation implementation.

• Chapter 5 is about the different performance tests which have been done to provide configuration recommendations.

• Chapter 6 introduces some search and index possible optimizations. Here it is discussed how an advanced search by name could be done using MongoDB.

• Chapter 7 summarizes the main results.
Chapter 2

State of the art and technical choices

2.1 Defining NoSQL

2.1.1 What is a database?

A database is a collection of data that is specially organized for rapid search and retrieval by a machine. Databases are done to easily create, read, update and modify data (CRUD operations) [2]. A database management system (DBMS) is the system which performs fast search or retrieval from a database. It also addresses issues such as consistency, memory requirement, latency and security [2].

2.1.2 What is a Relational Database Management System?

In 1970, Tedd Codd from IBM created the relational database model [3, 4]. With that model was also created an experimental database system called system R, and a language to query it called SQL (structured query language). In the relational model data are stored as a set of tables where a table has attributes (columns) and tuples (rows, records). Each table contains different information, for instance in the hotel use-case, it could be a table for customer information and another one describing customer’s room-stay(s). The table containing information about customers’ room-stay(s) could have in its attributes an ID defining a property, check-in date, and check-out date. Inside a tuple, data of a specific room-stay could be found. Since all data are related it is possible to access data from different tables at the same time. In RDBMS, we can find primary key and secondary key. Primary keys uniquely
identify tuples in a table. Foreign keys are used to cross-reference tables. A foreign key in one table represents the primary key in another one [5]. For instance, in the given example a link could be done between a customer and a room-stay.

### 2.1.3 Why do we need a new way to store our data 40 years later?

Nowadays the amount of data that a DBMS should manage has increased in an exponential way, and when the data volume is huge, relational systems become slow [5] and therefore applications start losing performance. RDBMS actually might not scale and might be not be able to support high demanding SLA (Service-Level Agreement). These are the reasons why it might be interesting to move toward NoSQL databases. Another reason NoSQL databases are emerging is because they provide a data model that fits better the application need (impedance mismatch) and result in less code to write and thus to debug [6]. The impedance mismatch already tried to be solved by Object Oriented databases which did not manage to be largely adopted. According to the same author another reason is that the majority of NoSQL databases are designed to run across several machines in different clusters unlike relational databases which are more designed to run on a single machine (see 2.1). It means that if you want to improve performance and scale with a relational system you will need to buy a bigger box. This approach is costing a lot for adding a little bit more performance unlike NoSQL databases where you will need to add several cheaper machines.

![Diagram: Vertical and Horizontal scalability](image.png)

**Figure 2.1: Vertical and Horizontal scalability**

### 2.1.4 Finally what does NoSQL mean?

The first time the NoSQL term appears was in the late 90’s by Carlo Strozzi [6] as the name of an open source relational database which did not use the SQL interface. Other than the name, this database did not have any impact on what we call NoSQL today [7].
The term NoSQL term comes from a meetup on June 11, 2009 in San Francisco organized by Johan Oskarsson. He wanted a name for this meetup and asked the question “What’s a good name” on IRC\textsuperscript{1}. He got a few suggestions and chooses the suggestion of NoSQL from Eric Evans who suggested this term “without thinking”. At that time they were just thinking of a name for the meeting and did not expect to name this technological trend [7, 8]. The original topic for this meetup was "open source, distributed, non-relational databases [9].

There is no official definition, but here are the characteristics NoSQL databases tend to follow [7]:

- Not using the relational model nor the SQL language. However some NoSQL databases offer a query language that is similar to SQL to make it easier to learn. We can mention Couchbase’s N1QL and Cassandra Query Language (CQL). It is also easy to convert MySQL query to MongoDB syntax [10]. However there is at the moment of writing no NoSQL database which implement the standard SQL.
- Open source (except some such as Amazon DynamoDB and SimpleDB).
- Designed to run on large clusters.
- Based on the needs of 21st century web properties.
- No schema, allowing fields to be added to any record without controls.

According to Eric Evans, MongoDB and Voldemort try to solve different problems and it is not meaningful to group them under the same NoSQL term [8]. But since they both try to solve problems that relational database are bad at, that might explain why some people interpret NoSQL as not relational database. According to Emil Eifrem (CEO of Neo4J), NoSQL does not mean “No To SQL” and it is simply Not Only SQL [12]. This definition tends to override the original definition [8]. However according to Martin Fowler interpreting NoSQL as "Not Only" is silly because it would render the term meaningless. Indeed he says that in that case we could argue that SQL server is a NoSQL database! That’s why he thinks that it is best to say "no-sql" database and suggest not to worry about what the acronym means, but instead in what it is standing for : “an ill defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL” [13]. On the other side the "not" can have its meaning if we refer to polyglot

\textsuperscript{1}Internet Relay Chat.
persistence, this is to say using different data stores for solving different issues in the same system. For instance we could build a system by mixing classical RDBMS and different NoSQL databases [7].

2.2 Relaxing consistency: from ACID to BASE

2.2.1 CAP Theorem

In a perfect world we would like a database management system to follow the Consistency, Availability and Partition tolerance [CAP] properties [14]. Consistency means that after an update or insert, users should all see the same data. If the same data is stored in several nodes, all nodes should always contain the same data. Therefore a consistent write is done, only if the data has been duplicated or updated to all nodes.

In an available system all requests are answered regardless crash or downtime. The system is always up. However availability has a particular meaning in the CAP context. It means that “every request received by a nonfailing node must result in a response” [7, p 54], if a node is available we should be able to read and write [7, 15].

A partition tolerant system can continue to operate, even if there is a communication breakage in the cluster, that separates it into unable to communicate multiple partitions (set of server that are isolated) [7].

Unfortunately in a distributed system it is impossible to simultaneously provide all of these three properties. Only two out of three of these properties can be guaranteed. This is known as the Brewer conjecture [16] or CAP theorem. The conjecture was made by Eric Brewer and was proved in 2002 by Seth Gilbert and Nancy Lynch, rendering the conjecture a theorem [7, 15].

In NoSQL databases we want to have P, so we need to select either C or A. If we drop A, we accept waiting until data is consistent. If we drop C, we accept getting inconsistent data sometimes (eventual consistency) [6]. For instance MongoDB is a CP whereas DynamoDB is AP. However Graph databases such as Neo4J, which are considered part of NoSQL movement are actually CA and do not partition [17]. In 2.2, some NoSQL databases are situated in the CAP theorem.
2.2.2 ACID properties

Relational databases follow ACID rules. ACID stands for:

- Atomicity: All of the operations in the transaction will complete, or none will [19].
- Consistency: The database will be in a consistent state when the transaction begins and ends [19].
- Isolation: The transaction will behave as if it is the only operation being performed upon the database [19].
- Durability: Upon completion of the transaction, the operation will not be reversed [19].

2.2.3 BASE properties

NoSQL databases have abandoned ACID rules and might follow BASE rules [14, 20]

- Basically available: an application works basically all the time (in spite of partial failure) [14, 20].
- Soft state: is in flux and non-deterministic [14, 20].
- Eventually consistent: will be consistent at some time in the future [14, 20].

According to Eric Brewer “ACID and BASE are two design philosophies at opposite ends of the consistency-availability spectrum” [21]. The ACID properties focus on
consistency and is the approach which has always been followed in databases, in particular relational before NoSQL trends. Brewer and his colleague introduced BASE so as to capture the emerging design approaches for high availability [21]. The original idea was that by giving up ACID rules, we can achieve better performances and scalability [22].

It should also be noticed that Graph databases which are attached to NoSQL trends actually follows ACID rules [22].

2.2.4 Is the “two of three” binary?

2.2.4.1 These properties are actually continuous

In reality these properties are more continuous than binary, in an article published twelve year later after PODC\textsuperscript{2} 2000, Eric Brewer explains that availability is actually continuous from 0 to 100 percent, and that there are also many levels of consistency, and even partitions have nuances [21].

In order to illustrate this point and also how NoSQL databases lose consistency to gain on availability and partition, we are going to describe how DynamoDB is working.

2.2.4.2 An example to understand CAP: DynamoDB

DynamoDB is the highly available-key value store from Amazon. In order to achieve a high level of availability, Dynamo trades off strong consistency [23].

Dynamo distributes and dynamically partitions the data on different nodes by using consistent hashing. Each node is randomly assigned a position in the ring (see figure 2.3), then each data item (the value) which is identified by a key is assigned to a node by applying an hashing function to the key. Therefore each node is responsible for a range of keys. The coordinator is responsible for this operation [23].

Also in order to provide high availability, Dynamo replicates values in several nodes. A value is replicated several times in different nodes. N is the number of times a value is replicated. That is why the coordinator copies values which are on a node at the N-1 clockwise successor nodes in the ring (see figure 2.3). As a consequence each node actually contains the value for the range of keys in the ring between it and its Nth predecessor. For instance in the figure below, node D is responsible for

\textsuperscript{2}Principles of Distributed Computing conference.
the range of keys (C, D], but if N = 2, B replicates the key k at nodes C and D, and C replicates its data to node D and E. That’s why node D will also store the keys that fall in the ranges (A, B], and (B, C] [23].

By applying the same reasoning to B and C, we can show that nodes B, C and D store keys in range (A,B) including K. The list of nodes that is storing a value associated to key is called the preference list. It should also be noticed that nodes can be physical, but is also possible to create virtual nodes (a physical node contains several virtual nodes) [23].

Dynamo is an eventually consistent system which allows updates to be propagated to all replicas asynchronously, a put() operation can be successful whereas the update has not been applied to all replicas. Therefore a get() operation can return an old version of the value. To know which version is the most recent, in a single node system it would be sufficient to use timestamps but this is not possible in a distributed system. That is why DynamoDB uses the vector clock algorithm so as to determine which version of the value is the most recent and also detect conflicts between two versions due to this eventual consistency. In case of conflict between two values, a conflict resolution is done. At Amazon a common example is the shopping cart, where the conflict resolution is done by merging the two shopping carts in conflict. Therefore "add to cart" operations are never lost, however deleted articles can reappear [23].

To maintain consistency among its replicas Dynamo uses a consistency protocol close to the one used in quorum distributed systems. The system has two values, which can be set : R and W. R is the number of nodes that must be involved for a successful read operation, and W is the number of nodes that must participate in a successful write operation. According to the author, if R + W > N, it yields to a

\[ \text{In distributed system, a quorum is the minimal number of nodes in the network which are sufficient to make a decision} \]
quorum like system. A quorum system guarantees serial consistency. In this model latency in read and write is defined by the slowest node, for this reason so as to improve latency we often choose \( R < N \) and \( W < N \) (then if \( R + W < N \) consistency will not be guaranteed). When a put request is received by the system a vector clock is generated and the new version of data is written locally. Then the coordinator replicates the value to \( N \) nodes. If at least \( W - 1 \) nodes respond we consider the write operation is successful. For a get() request, all existing versions of the value for that key from the \( N \) nodes are requested and then the system waits for \( R \) responses before returning the result to the client. If different versions are detected by the vector algorithm, data are reconciled and written back on the system [23]. This approach is statistic and we should notice that conflict resolution is done at reading time instead of writing time [24].

2.2.4.3 What we learn from the example

What we can see here is that \( R, W \) and \( N \) determine our level of consistency and availability and situates us in the CAP theorem. If \( R = W = N \) our system is highly consistent, but we lose in availability since we need to request all nodes. On the other hand if \( R \ll N \) and \( W \ll N \), consistency will be low but high availability will be provided by the system [25].

Therefore if DynamoDB is told to be an AP key-value store what can say here is that it can become consistent depending on the value we choose for \( R, W \) and \( N \) and be closer to a CP system.

2.3 Different families of NoSQL databases

There are over 150 NoSQL databases [11] which can be grouped in four different categories:

- **Key-value store**: All data is stored as a set of keys and their values. The key is unique and data can be uniquely accessed by its key. There is no way for the database to access the content of the value [5] and the only operation we can do is a retrieve of a document by its key. The value is a black-box. Examples are DynamoDB, Riak, Redis.

- **Document store**: they are similar to key-value stores except that a document is defined in a known standard such as XML or JSON. In a document store, the
database can access of the content of the document, in particular it can find a document according to its content, and can create indexes on the document fields (secondary indexes) [5], and retrieve only some part of the document [7, p 20]. The value is a white-box. Examples are MongoDB and Couchbase

- **Column-family store:** they are the most similar to relational databases. Data is structured in columns. They store data in a column family as a row. This row has a row key and many columns associated. This associated column is the unit of data identified by a key and value [5, 7]. Examples are Big Table, HBase, Cassandra.

- **Graph database:** they are used to represent data as a graph (nodes, edges) such as a social network. They store entities and relationships between these entities. Graph databases are very powerful to manipulate connected data or even creating recommendation engines [5, 7]. Examples are Neo4J, OrientDB.

- **Some authors also include object oriented database systems and distributed object oriented stores in the NoSQL trend** [22]. These two databases and Graph databases actually follow ACID rules [22].

It should be noticed that as in all classifications there are some limitations, for instance OrientDB is often considered as a graph database, but it can also be seen as a document database. Indeed OrientDB stores JSON documents (nodes level) therefore as a document store, and it connects them by using "direct, super-fast links taken from the Graph Database world" [26]. Couchbase can also store a binary instead of a structured JSON document, in that specific configuration Couchbase will work as document store. Also there are no strong links between the different NoSQL families and their relation to the CAP theorem. HBase is CP column store whereas Cassandra is an AP column store [27]. In figure 2.4, several examples of NoSQL databases are given with their classification in each of the categories defined below.

### 2.4 Current solution for storing hotel reservations

#### 2.4.1 e-voucher image

All information regarding a booking is stored in an electronic folder called an e-voucher. Each sell, modify or cancel transaction creates what is called an e-voucher image (EVR) and is included it in the e-voucher. Therefore an e-voucher contains
several e-voucher images, and the latest image (higher version number) represents
the current state of the reservation.

The EVR (e-voucher image) is a Protocol Buffers (similar to JSON) document which
contains different information regarding the booking IDs, customer details, form of
payment, room-stay, full pricing and policy, amounts ... The EVR acts as contract
between the customer and the hotel. More details are given in 4.1.2.

2.4.2 Current storing solution

The current system contains a set of Oracle tables. An e-voucher image is stored as
a blob. Some content of this blob is replicated to columns so as to be able to make
a search or retrieve of booking.

2.5 Choice of a kind of NoSQL database

The blob which contains the e-voucher is already a structured document therefore it
sounds natural to choose a document store for this project. A simple key-value store
would not be sufficient since it would not enable to provide search functionality
(search by name and date for instance). A column family store could work well but
would need a different data modeling from the current one. Graph databases are
here not necessary since we do not need to model relationship between different
documents. In the next chapter the two most popular document stores, MongoDB
and Couchbase will be studied.
Chapter 3

Document store

In this chapter MongoDB and Couchbase are compared so as to highlight differences and similarities between these two document stores. This part of the work was fundamental to weigh the pros and cons on these two technologies and select the one which will help to do the best we want to achieve. First we compare them from a functionality perspective. For this purpose a Python program doing search, insert and retrieve operations using a MongoDB or a Couchbase strategy has been written using the Python library of these two databases. Then their architectures are studied. This comparison enables to make a decision on which document database seems to fit the best for the proof of concept.

3.1 Couchbase and MongoDB overview

Both MongoDB and Couchbase are a CP (if we refer to CAP theorem) document stores.

3.1.1 History

MongoDB is developed by the 10gen (now MongoDB Inc.) company and was released first in 2009. Among MongoDB customers we can mention companies such as EBay, Sourceforge, The New York times.

Couchbase server is developed by Couchbase Company and was released first in 2011. Some of their customers are AOL, Cisco, Linkedin, Zynga(wiki). Historically Apache CouchDB (a document store) and Membase (a key-value store) projects merged to form Couchbase [28]. Couchbase engineers started with Membase as the
base technology, and reused certain aspects of CouchDB code to replace the Membase storage back-end, indexing and querying functionality (document functionalities) [29]. The Membase project was developed by the leader of the Memcached project (which was an in memory key-value store [30]) to build a Memcached with persistence and querying capabilities.

According to some recent benchmarks Couchbase has better performance than MongoDB for the retrieve operation [31, 32].

### 3.1.2 Kind of database and the value they store

Couchbase and MongoDB as a document store have both access to the fields of the object (document) they store.

#### 3.1.2.1 MongoDB: A full document store

MongoDB stores the value as a BSON (Binary JSON) document. BSON is a superset of JSON which contains types such as date or ObjectID. The maximum size of BSON documents MongoDB can store is 16MB. It is also possible to store bigger files or binary files using the GridFS functionality [33, Data Models], MongoDB will in that case split the file or binary into chunks of 255 KB [34, Data Models].

MongoDB documents are stored in what is called a collection. A MongoDB instance can have one or several databases and each database can contain one or multiple collections (figure 3.1). The collection is equivalent to a table and the value to a row in the relational world.

![Figure 3.1: MongoDB data model](image)

In MongoDB, the _id (key) is usually generated by the database but can be specified by the client. The key is contained in the document itself. MongoDB is a full document store, it enables to retrieve document by the primary key, the _id field and make search of documents based on fields different from the _id (key), and offers the possibility to index those fields (secondary index).
3.1.2.2 **Couchbase: From a key-value store to a document store**

Couchbase stores objects with a maximum size of 20MB [35]. The objects stored by Couchbase can be any binary value including native binary data (such as images or audio). To use Couchbase document features (Couchbase Server View engine) information must be stored using JSON document [36]. Otherwise Couchbase will work as a key-value store.

Couchbase does not have the collection concept and stores all documents in what is called a bucket. Thus if we want to have two collections, we can either create two buckets or one bucket with adding a type field containing the document type in the document itself.

As it was highlighted in the history subsection (3.1.1), Couchbase is actually a key-value store which turned into a document store. This might explain, as we will see later why document functionalities are not as mature as in MongoDB. The key (equivalent to MongoDB id field) is not contained in the document itself but in what is called document metadata. To retrieve a document by its key, the get function is used. To search a document by a document field (not the key) a different mechanism called 'views' is used (secondary indexes).

3.1.2.3 **Both are schemaless**

Both JSON value in Couchbase and MongoDB are schemaless, this means it makes it possible to store documents with different fields name or having a document with more fields in the same collection or bucket. However this might lead to some issues. For instance if we insert into the database the two following documents

1. `{"lastName": "spielberg", "firstName": "steven"}`
2. `{"Name": "allen", "firstName": "woody"}`

There won't be any problem at insertion time, but if your queries were built to search documents which lastName field is equal to a value, you will also have to modify your query and make a "or" on Name and lastName fields to have something consistent. Note you will not receive any error from the database and that there is no need with Couchbase and MongoDB to define a schema (such as CREATE TABLE in SQL) before inserting documents.
3.2 Couchbase and MongoDB functionality study

3.2.1 Database connection

3.2.1.1 Couchbase

Couchbase establishes a connection to a bucket. We will call the connection object c.

```python
self.c = Couchbase.connect(bucket='default', host='localhost')
```

3.2.1.2 MongoDB

The MongoDB connection process is similar to the Couchbase one, lines 2 and 3, enable to select the database and the collection we want to work on.

```python
c = MongoClient()
sel/db = c.db
self.coll = sel/db.bookingCollection
```

3.2.2 Document insertion

For comparing the two databases we will fill a Couchbase bucket and a MongoDB collection with the following documents:

```python
1 {"name": "codd", "cardNum": "341", "hc": "AX", "cc": "A", "n": 3},
2 {"name": "codd", "cardNum": "341", "hc": "AX", "cc": "A", "n": 2},
3 {"name": "codd", "cardNum": "341", "hc": "BX", "cc": "B", "n": 2},
4 {"name": "coulombel", "cardNum": "351", "hc": "AX", "cc": "A", "n": 2},
5 {"name": "coulombel", "cardNum": "351", "hc": "AX", "cc": "A", "n": 1},
6 {"name": "coulombel", "cardNum": "BY", "cc": "B", "n": 2},
7 {"name": "weiser", "cardNum": "361", "hc": "AY", "cc": "A", "n": 1}
```

The document contains a customer’s name, the loyalty card number, a hotel code identifying a specific hotel and a chain code which identifies a set of hotels, n is the number of nights spent during the room-stay.

3.2.2.1 Couchbase set() method

To insert a booking (a document) called b into the database, Couchbase will use the set function which takes as parameter the key and the JSON document to insert.

```python
self.c.set(key, b)
```
3.2.2.2 MongoDB insert method

MongoDB uses the insert() method, the id (key) is automatically generated but can be specified directly in b.

```python
bookingID = self.coll.insert(b)
```

3.2.3 Document retrieve

3.2.3.1 Couchbase get() method

Couchbase document retrieve by key is done using the get() function:

```python
doc = c.get(key)
```

3.2.3.2 MongoDB find_one() method

In MongoDB it is done using the find_one() method applied to a specific collection. The find_one function takes as a parameter a JSON document (dictionary in Python) which represents the query.

```python
doc = self.coll.find_one("
```

3.2.3.3 A functional retrieve (on a different field from key or id)

It is also possible with MongoDB to apply find_one() method on secondary indexes and create unique index on a secondary field (a field which is not the _id). Therefore it makes it possible to implement a retrieve functionality on a secondary index. In hotel use-case the _id could be a booking identifier generated by the database. In that case it we will be possible to make a retrieve by confirmation number (CF):

```python
doc = self.coll.find_one("
```

Therefore it makes it possible to implement a retrieve functionality independent from the document key (_id field) itself by creating an index on "CFNUMFIELD". It will also possible to add a unique constraint on the index. Actually in MongoDB the id (key) field is an ordinary one (except it can not contain an array) which is

---

1 In Python API, equivalent to findOne() in native JavaScript (JS) API.
2 However we should pay attention to the choice of the shard key in that case. The shard key concept in MongoDB is explained in 3.3.1.2.
contained in the document and is always indexed [34]. In Couchbase this is not possible as the primary index is managed in a different way as the secondary indexes. The primary key is in the metadata and secondary indexes are contained in views which are often asynchronous actually. One solution to be able to implement a functional retrieve would be to make a double indirection (the original document with keyO and a second document with the confirmation number as a key and keyO in value, therefore to get a document by confirmation number we need to make two retrieves).

### 3.2.4 Document search

We can search for documents whose field are equal to a value, for instance one such query could be to find all the documents where the name starts with a specific letter. MongoDB and Couchbase approaches are very different.

#### 3.2.4.1 MongoDB `find()` method

MongoDB uses the `find()` function. The find function takes as a first parameter a query document (JSON document and a dictionary in Python API), by adding key/value pair in the query document we restrict the search. For instance to find all documents where the name is coulombel the query document will look like `{"name": "coulombel"}`. It returns a cursor to iterate on to get all results. Here is an example of a search of documents of all names starting with a letter.

```python
queryDict = {"name" : {"$regex" : '^' + letter}}
for booking in self.coll.find(queryDict):
    print(booking)
```

The find function can take as a parameter a second document, which represents the field we want to project. The first document is equivalent to the WHERE clause in SQL and the second one to the SELECT clause. It is also possible to limit the number of documents to return, sort them by a specific or a combination of several fields which can be the same or different from those used in the search. The following example will only display the first three results and only shows customer names (and the id), and will sort the result by customer’s name and hotelCode.

```python
queryDict = {"name" : {"$regex" : '^' + letter}}
projDict = {"name":1}
cursor = self.coll.find(queryDict, projDict).limit(3).sort([("name" , -1) , ("hotelCode" ,1)])
```

3`collection.find_one(queryDict) method is equivalent to get the unique value from collection.find(queryDict).limit(1).pretty(), find_one is therefore not limited to retrieval.`
If the number of documents in the database is huge, it can be interesting to avoid a full document scan and create an index here on the name field. We discuss indexing and sort parameter in 4.5 and 6.1.

### 3.2.4.2 Couchbase views

Couchbase uses the map-reduce\(^4\) framework even to make simple search. First we need to define on server side, a map and a reduce function called a view in Couchbase terminology. The view defines at the same time the query and index, which are created on fields used in the emit function parameters inside the map function. For a simple search we do not need the reduce function. Then this view is called from the client. Therefore in order to be able to perform the same query as we did before in MongoDB (first one) we need to define a view which is JavaScript map function:

```javascript
function (doc, meta) {
    emit(doc.name, doc);
}
```

We emit a key and value. The output of map function will be\(^5\):

```javascript
{"key":"codd","value":{"cc":"A","hc":"AX","n":3,"name":"codd","cardNum":"341"}},
{"key":"codd","value":{"cc":"A","hc":"AX","n":2,"name":"codd","cardNum":"341"}},
{"key":"codd","value":{"cc":"B","hc":"BX","n":2,"name":"codd","cardNum":"341"}},
{"key":"coulombel","value":{"cc":"B","hc":"BV","n":2,"name":"coulombel","cardNum":"351"}},
{"key":"coulombel","value":{"cc":"A","hc":"AX","n":1,"name":"coulombel","cardNum":"351"}},
{"key":"coulombel","value":{"cc":"A","hc":"AX","n":2,"name":"coulombel","cardNum":"351"}},
{"key":"weiser","value":{"cc":"A","hc":"AY","n":1,"name":"weiser","cardNum":"361"}}
```

The document key (which was emitted in the map) are now the names.

\(^4\)Map reduce is a way of computing which enables to manage large scale computation across several machines and that is tolerant to hardware failure. From the programmer point, it is only needed to write a map and a reduce function. Map-reduce examples are shown in this section and 3.2.5.2. For more details we can refer to [37].

\(^5\)the original document key is also in the map output in reality (in an id field), we removed it for simplicity reason.
This view is called from the Python client, so as to only select the document where the name start by a specific letter we add start and end key constraints. The View function returns a cursor with the result. The results contains the same document as those in the map function output except the document which customer’s name is "weiser".

In the example, inside map function we made an emit(doc.name, doc), therefore the value is the whole document. This is not recommended by Couchbase as it means we copy all the documents in the database to the index. Therefore it is recommended to emit only the values which are really needed. For instance like this emit(doc.name, doc.hotelCode) or if there are several fields, we can emit the value in an array like this emit(doc.name,[doc.chainCode, doc.hotelCode]). Similarly the emitted key can also be an array. As in MongoDB it is possible to limit and skip results however if we can sort the result by ascending or descending emitted key, it is not possible to sort by the emitted value or another field.

### 3.2.4.3 Comparison

It sounds to us more natural to write complex queries with MongoDB than with Couchbase views. Also it appears that in MongoDB we first define the index and then make query which can use the index or not. Thus MongoDB separates index creation from query definition. When we create a Couchbase view, we are actually defining at the same time indexes and a part of the query. Therefore it is less flexible than MongoDB since we need to index all the fields emitted by the map-reduce operation. MongoDB enables to make better index optimization, such as using the same index for two queries, or not using index if the query is only used by DBA, or scan almost the entire collection.

### 3.2.5 Document aggregation

Aggregation enables to analyze data, gather information from different documents and rearrange data in an interesting way. For instance it enables to answer requests

---

6If the whole document is needed, a better solution would be to emit the key, and then retrieve the document using the `get()` method.

7Since using the MongoDB `find()` function is more convenient than writing a Couchbase view (map-reduce) on server side and calling it from the client.
such as "within chain code A, give me the number of nights spent in a particular hotel for a given loyalty card number, and sort results by numbers of nights".

### 3.2.5.1 MongoDB aggregate method

This is done easily in MongoDB with the aggregation framework like this:

```python
res = self.coll.aggregate(
    [
        # select only booking in chain code A
        {"$match": {"cc": "A"}},
        # group by hotel code and card number
        {"$group":
            "_id": {"hotel": "$hc", "cardNum": "$cardNum"},
            # count the number of booking made
            "totalStay": {"$sum": "$n"}
        },
        # sort by number of bookings
        {"$sort": {"totalStay": -1}}
    ]
)
```

### 3.2.5.2 Couchbase views

Couchbase will use again a map-reduce operation, map and reduce functions are given below:

```javascript
function (doc, meta) {
    if(doc.cc == "A"){
        emit([doc.hc, doc.cardNum], doc.n);
    }
}

function(key, values, rereduce) {
    var count = 0
    for(i=0; i < values.length; i++) {
        count += values[i];
    }
}
```

The `map()` function starts in the if condition by filtering all documents in chain code A. Then for each document emit the hotel code and the card number such as the output of the `map()` function is:
The map-reduce framework will gather the documents with the same key as follows:

1. \{key: ["AX", "341"], value: 3\},
2. \{key: ["AX", "341"], value: 2\},
3. \{key: ["AX", "351"], value: 1\},
4. \{key: ["AX", "351"], value: 2\},
5. \{key: ["AY", "361"], value: 1\}

For each unique key the \textit{reduce} function is applied, here we sum the content of the values array, so after the \texttt{reduce()} we get:

1. \{key: ["AX", "341"], value: 5\},
2. \{key: ["AX", "351"], values: [1, 2]\},
3. \{key: ["AY", "361"], value: 1\},

We got the same results using MongoDB and the aggregation framework. This map-reduce functions are called from the Python client in a similar way as for search purpose\(^8\). There is no way to sort map-reduce results as we did with the aggregation framework in MongoDB.

### 3.2.5.3 MongoDB Aggregation framework limitation

Sometimes in MongoDB the aggregation framework is not sufficient, when we want to make complex queries that are not possible to be expressed with the aggregation framework [38] (for instance not using basic operations such as min, max, average or sum on a document field). Let’s suppose we want to sum the number of letters of all customers name by property (this is just for the example), this would require using map-reduce function of MongoDB (in Couchbase it would obviously remain a map-reduce). When MongoDB performs map-reduce, it outputs results in a collection which enables to manipulate them easily. When possible it is better to use the aggregation framework as the map-reduce performs slower than the aggregation framework [38]. In particular index can not be used in the map and reduce part but we can use query option (which can use index) [33, Reference] to avoid to pass all documents of the collection in the map and reduce functions\(^9\).

\(^8\)The group level is here equal to 2 as the emitted key is the full array [doc.hotelCode, doc.cardNum], since we want to group by this two fields.

\(^9\)Aggregation framework can use index in particular the $match operator, thus is recommended to use this operator at the beginning of the aggregation pipeline in particular before renaming field which will make the index useless, it will also make less documents to be processed by the aggregation framework.
Both Couchbase and MongoDB have a map-reduce but they are not used for the same goal at all. Couchbase is currently developing a language called N1QL which enables to write Couchbase views using a language similar to SQL.

### 3.2.6 Special indexes

MongoDB officially supports from 2.6 text indexes (full-text search), this can be really useful for the search by name functionality and Geospatial index. Couchbase has geospatial index (geocouch) but does not support them officially yet. It also does not have text index embedded unlike MongoDB (but it is possible to integrate Couchbase with ElasticSearch to provide this functionality).

### 3.3 Couchbase and MongoDB Architecture

We compared Couchbase and MongoDB in terms of functionalities. We will now focus on architecture. There are two ways to distribute data [7, p 44-45].

1. **Sharding** splits data and distributes it into multiple nodes (machines), so each node will contain a subset of data [7]. This is how NoSQL databases scale vertically. Instead of scaling horizontally by having more powerful machines (more RAM, better CPU), we add more machines. NoSQL solutions are usually designed for this kind of scalability. It enables to reduce the load on a single machine, by distributing the workload and reduce the amount of data a machine has to store.

2. **Replication** copies data to multiple nodes, the same part of data will be contained in different nodes [7]. This is how NoSQL databases provide high availability. Replication can be used to (in parenthesis we will give concepts it is referring to in MongoDB and Couchbase context, this concept being developed below):

   (a) recover from hardware failure and service interruption (MongoDB replica set, Couchbase intra and extra cluster replication concept).

   (b) provide high availability by increasing read capacity. It is also possible to maintain copies in different data-centers to increase locality and availability (MongoDB replica set with reading activated on secondary, Couchbase extra-cluster replication: XDCR\(^\text{10}\)).

\(^\text{10}\)It is also possible in a cluster (intra-cluster) to read on replicas
Replication can have two approaches:

(a) master-slave: one node is the master node which contains the reliable copy and receives all reads and writes, whereas slave nodes replicate from the master. Slaves might also be to answer read requests but not write [7] (MongoDB replica set, Couchbase intra-cluster replication concept (read activated or not)).

(b) peer-to-peer replication: all nodes can receive write, and nodes coordinate to synchronize their copy of data [7] (In a way¹¹ Couchbase extra-cluster replication: XDCR, does not exist in MongoDB).

Note that replication is different from sharding. However replication and sharding techniques can be combined. MongoDB and Couchbase fully use this techniques.

### 3.3.1 MongoDB architecture

#### 3.3.1.1 Replication

![Figure 3.2: MongoDB: a replica set](image)

Source: MongoDB documentation [33]

¹¹In Couchbase it is more "a node in a data-center can receive a write and coordinate with another data-center", a better example of this kind of replication is the one used in Cassandra.
Read and write  Figure 3.2 is an illustration of a replica set. The primary (master, in blue) receives all reads and writes and replicates them to secondaries (slaves/replicas, in green). The replica set can also work without data replication and the two secondaries. It is also possible to replicate a secondary from a secondary (chained replication) [33, replication]. Replication uses oplog [33, replication], a special collection which contains every operation made by the primary [38, p 187]. All replica set members contain a copy of the oplog. We can decide if replicas should build an index or not [38, p 222]. In terms of implementation a primary or a secondary is a process accessing a data file. This process is called a MongoDB.

By default a client reads from primary, however it is possible to allow clients to read read from secondaries [38, p 44-45] by changing read preferences. However it will not be possible to allow clients to write directly on secondaries. When allowing reading on secondaries if an insert or update was done one the master but the secondary we are reading from did not have time to replicate the data from the master, the client will read stale data. This is again a good illustration of CAP theorem, by allowing read on secondaries, we got an higher availability, but we pay it in consistency. Note that slave nodes (replicas) can be located on different data-centers [7, p 39], therefore allowing reading on secondary can be useful for low latency read by making the client read data from the nearest data-center [33, replication] (note that however write will always be from the master).

Failover  In case of node failure, a secondary server is elected as a primary. More exactly when a secondary detects it can not reach the primary, he contacts other members of the set and asks to become a primary. The member seeking to be elected must have replicated the data as far as the other members in the replica [38, p 180-181]. If the member receives a positive vote from the majority of the members in the set it will become the master node. If one node in the replica has a good reason to prevent the election, it can veto and cancel the election[38, p 180-181]. For instance a node in the replica set might have replicated more operations than the member seeking for election and can veto it. Then the member who had replicated more operations and vetoes the election might become the new primary or the member who had its election canceled could replicate the latest operations ask again for election [38, p 180-181]. If the master node recovers later it will join as a slave node [33, sharding].

Also in order to be able to elect the new primary more than a strict majority of the node in the replica set should be available or reachable by the member seeking to be elected. Moreover if a master node realizes it can not reach the majority of nodes
in the replica set, it will step down and become a secondary. This conception choice
is done to avoid to elect two primary at the same time [38, p 180-181]. For instance
it there were 7 nodes in the set where 4 nodes are in the European data-center and
3 nodes in the US data-center and if there was a transatlantic network outage, this
prevents MongoDB from electing a master in US and another one in Europe [38,
p 180-181]. This has two main consequences when designing a replica set:

1. A two-member replica set (master slave) is not recommended, since if one
node fails the number of nodes in the set is 1 whereas the strict majority is 2.
   Therefore the secondary won’t be elected as a master.

2. If replica sets are distributed across several data-centers, a majority of nodes
   should be put in one data-center so as to be able to elect a primary in case
   of network outage between the two data-centers [38, p 181] or put the same
   number of nodes in two data-centers plus another server in a third data-center
   [38, p 181].

**Write concern**  MongoDB has different consistency levels for write operations in
a replica set.

1. Unacknowledged write concern: MongoDB does not acknowledge the receipt
   of write operations.

2. Acknowledged: MongoDB (which is the process managing a primary or sec-
   ondary) confirms the receipt (returns hand to the client) of the write operation
   (catch also errors such as duplicate keys).

3. Replica acknowledged: MongoDB guarantees that the write operation propa-
   gates to additional nodes of the replica set.

We set \( w \) to the number of secondaries we want to replicate, \{\( w:1 \}\} is the default
value and corresponds to an acknowledge write\(^{12}\). If we set \{\( w:3 \}\}, the client will
return the hand after the data was replicated to the master and at least two replicas.
We can use \( w:\text{majority} \) to replicate to the majority of nodes within the replica before
returning the hand to the client. By default \{\( w:1 \}\}, so write is only acknowledged
by the master [33, replication]. It is recommended to choose a write concern of at
least majority (so returning hand to the client when the data has been replicated
to the majority of nodes), because in that way the node which will be elected as a
new primary will contain for sure the latest updates (as we said that the member

\(^{12}\)In latest driver, from Pymongo 2.4 in the case of Python.
seeking to be elected must have replicated the data as far as the other members in the replica). If we don’t replicate to the majority of nodes within the replica set, if the master node fails, data written to the master might not be replicated to the new elected master. Therefore when the former master node will come back into the replica set, it will start replicating from the new master and will realize it has data which are not in the current master. The former master will rollback these operations and put them in a file. The database administrator will be able to restore these documents manually on the new master [38, p 193-194]. This is usually something we want to avoid.

**Journaling** Each MongoDB has a data file on disk, when we start a MongoDB it maps this data file to RAM. The data is first written to RAM and then committed to disk. It has not been explained yet what happens between these two steps. MongoDB has a mechanism called journaling which we can activate or not.

When no journaling is activated on the server side (this is not recommended by MongoDB)\(^{13}\):

1. If the client requests a journaled write it will output an error\(^{14}\).

2. If the client requests a non journaled write (insert/update), change will be made on RAM, then the OS will flush changes to data file every 60 seconds [39].

If journaling is activated on MongoDB\(^{15}\). MongoDB will use a journal file.

When the journal was not used MongoDB only used the shared view in RAM and flushes its content to disk every 60 second [39]. When journaling is activated MongoDB uses also a private view in RAM and journal file in disk as shown in figure 3.3. When a write is done it writes the data to the private view which then writes to the journal file the modification (in which file which part of the data was changed) every commit interval [39]. At that moment our write is safe on the master node since we will be able to replay this write from the journal file if the node shut down [39]. Then the MongoDB replays the operation on the shared view by reading the journal file. And finally the shared view is committed to disk every 60s as before, except that if there is a node failure data which were in RAM and were not committed to disk will be safe if they have been committed to the journal [39].

\(^{13}\)Specify we do not want journaling by starting MongoDB with the nojournal option (default on 32 bits version).

\(^{14}\)Before 2.6 version the j option will be ignored.

\(^{15}\)Specify we want journaling by starting MongoDB with the journal option (default on 64 bits version).
1. If the client requests a journaled write, MongoDB will acknowledge write in the master node only after committing data into journal, also so as to make the client not to wait too long for the write it will reduce the commit interval.

2. If the client requests a non journaled write, MongoDB will use the journal[^16] and acknowledge write from the master node without waiting the data to be committed to the journal. Write will be faster as the commit interval will not be reduced but it will not be safe.

The journal is committed to disk every commit interval. This commit interval can be set [33, reference] between 2 ms and 300 ms (whereas data files are committed every 60 seconds). Note that a write operation is considered as durable in a replica set only after a write replicates and commits to the journal (or disk when journaling is not activated) of a majority of the members of the set [33, CRUD Operations]. If we use `{w:majority}` and `{j:true}`, we won’t have the guarantee that the write was committed to the journal of the secondary, but what MongoDB guarantees is that the master acknowledges the write after committing data to the journal, and that secondaries acknowledge the write without necessarily having it committed to the journal. This is in fact not a problem since write is safe with the journal of the master node, then if the primary fails and write was not replicated to the journal of secondaries, either the majority of nodes is still alive and therefore the write exists or it will not be possible to elect a new master [40, 41].

[^16]: This is not written in the current MongoDB documentation, we made the experience and noticed journal files were generated.
Read isolation  
MongoDB enables clients to read documents inserted or modified before the modifications have been persisted to disk. This behavior is independent from write concern or journaling [33, CRUD Operations]. Therefore a document is available for search as soon as it has been inserted.

3.3.1.2 Sharding

Architecture  
MongoDB can distribute a collection within different machines by putting a subset of data on different shards. A shard is an independent database, which by putting them all together forms the logical database [33, Sharding]. In that case the architecture of the cluster is the one shown in figure 3.4.

![MongoDB sharded cluster](image)

In figure 3.4, the cluster contains n shards. Each shards manages roughly 100/n percent of the data. If n=4, one shard will contain 25 percent of the data. As shown in figure 3.4 in blue, a shard is actually a replica set 18, seen in 3.3.1.1 section. Thus if a primary has secondaries, we combine master-slave replication and sharding technique. The configuration server contains metadata information necessary to make sharding work. If all members of a replica set become unavailable in a shard, all data in that shard will become unavailable, but data in the other shards will remain available [33, Sharding]. Each shard manages its own index [42].

Query routing  
If sharding increases the database scalability, it also brings its own set of problems. Assume you want to read or write data to your cluster, when

---

17 Having a number of shards n < 3 is not recommended as the sharding cost will be higher than the benefit of sharding.

18 Terminology reminder: 1 shard = 1 replica set = 1 primary + n secondaries; primary = master; secondary = slave = replica; machine = node; primary and secondary are managed by a process called a MongoDB.
there is only one replica set or one node, the process is trivial since the client will contact directly the good MongoDB as there is only one. If the collection is sharded then the interesting data might be found on one of the shards, or even in p shards. The database hides this complexity to the client, with a router (MongoS process). The router keeps a "table of content" and tells which shards contain which data. The router can therefore send the request to the appropriate shard or if responses are actually located on several shards, the router will merge the results from the different shards. Then the MongoS will send the result back to the client.

Sharding is done by collection, thus in a sharded cluster containing two collections, we can decide to shard only one of them. In that case the non-sharded collection will be located in a random shard (primary partition), and will also be accessible from MongoS.

In a production cluster we should have always more than one MongoS running on different machines. Indeed if there is only one MongoS we have a single point of failure. MongoS usually runs on the same machine where shards are hosted.

**Kind of shard key** When sharding a collection, we need to choose a shard key. This is an indexed field or an indexed compound field which enables MongoDB to split the data [33, Sharding]. MongoDB support two kinds of sharding.

1. Range-based partitioning: Data are divided into ranges determined by a shard key. Keys which are closer will be more likely in the same chunk [33, sharding]. Range-based partitioning is very efficient for range query (see 3.3.1.2) but can result in an imbalanced cluster [33, sharding]. For instance if we shard on an ascending field such as the transaction date, all the writes will go to a single shard (and also all queries in our use-case because last bookings are more active than former one).

2. Hash-based partitioning: MongoDB can also use hash-based partitioning where MongoDB creates chunks based on the hash of a field value. Unlike range-based partitioning, documents with close shard keys will be probably not in the same chunk. The distribution of data will be more random [33, sharding]. Hash-based partitioning, guarantees a well balanced cluster but range queries will be less efficient [33, sharding].

MongoDB also offers Tag-Aware Sharding which enables to control balancing strategy by creating and associating tags with ranges of the shard key, and assign a tag
to a specific shard. Then, given the tag, a balancer migrates tagged data to the appropriate shards [33, sharding].

The choice of a good shard key is critical, not easy and dependent on the application use-case. The choice of bad shard key will impact cluster performances, moreover it will be very difficult to change it (almost impossible) in case of wrong decisions.

### Data splitting

Based on the shard key the data is split into chunks. For instance if the chosen shard key is the customers last name field (range-based partitioning), chunk will be break up on range of last name, then these chunks are distributed in the different shards [38, p 235] as shown in 3.1.

When a chunk has reached a threshold (important number of data inserted in one chunk), MongoS will split the chunk into two chunks. Two documents with the same shard key should however be in the same chunk, for instance if the chunk contains only customers with the same last name, the chunk could not be split. That’s why it is important to have a variety of values in the chosen shard key [38, p 249-250]. A balancer is responsible for migrating chunks between shards, if data in the cluster is unbalanced (a node having more data than an other one).

### Query isolation

Query isolation is an important concept to consider when choosing the shard key. The idea is that the shard key could help the MongoS to determine directly the good shard(s) to target. For instance suppose we still have our sharded collection on customers last name field and we are searching all reservations made by customers whose last name starts with letter "c" and whose first name starts with letter "s". When the MongoS will receive this query, it will be able to determine directly on which shard(s) interesting documents are in this range (shard 1 in the example). If my query was a search of all the documents made by customers whose last name starts with "s" only, the query will be less efficient, as the MongoS will have to check in all shards (broadcast), if there is no first name starting with "s". Thus if in our query set, a field is contained in almost all queries it makes it a good candidate for the choice of the shard key.
3.3.1.3 Process and physical machine

In a sharded cluster we have three different components:

1. **MongoD** which manages each shard. Being a MongoD or a Primary (P) or Secondary (S).

2. **MongoS** which routes queries to MongoD and sends back the answers to the client. In a production cluster it is recommended to run several MongoS. If there is only one MongoS running we have a single point of failure. They usually run on application server.

3. **Config server** which contains cluster metadata. A production cluster has exactly three configuration servers (processes, for test purpose one can be sufficient).

Note that each of these components is a process and each of them is not necessarily running on a dedicated machine (in particular MongoS and Config server which are light weight processes) thus a machine can host several processes. Also in a cluster several instances of these components can exist.

For instance if we have a cluster with three shards A, B and C where each one is replicated twice, we can put on three machines a configuration similar to the one in table 3.2.

<table>
<thead>
<tr>
<th>Machine</th>
<th>MongoD</th>
<th>MongoD</th>
<th>MongoD</th>
<th>Config</th>
<th>MongoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>MongoD shard A P</td>
<td>MongoD shard B S</td>
<td>MongoD shard C S</td>
<td>Config U</td>
<td>MongoS 1</td>
</tr>
<tr>
<td>Y</td>
<td>MongoD shard A S</td>
<td>MongoD shard B P</td>
<td>MongoD shard C S</td>
<td>Config V</td>
<td>MongoS 2</td>
</tr>
<tr>
<td>Z</td>
<td>MongoD shard A S</td>
<td>MongoD shard B P</td>
<td>MongoD shard C P</td>
<td>Config W</td>
<td>MongoS 3</td>
</tr>
</tbody>
</table>

Table 3.2: Three replica sets on three machines

If node Y is down, one of the secondary of the shard B in physical node X or Z will be elected as master. Therefore that node will receive twice more loads (at least in write if we enable reading from secondaries) than the other one and might not support the load. Also note it would make no sense to put the three replicas of the same shard on the same machine. This configuration is a little crowded and is not really recommended in production. Indeed this configuration could lead at the end with the three primaries on one physical machine\(^\text{19}\)[43] (since former primary join cluster as a secondary when they come back). However in some situation it can be interesting to put several shards (MongoD) working as primary (not primary of

\(^\text{19}\)But it can be avoided with replica set priority, actually this setup mimics the Couchbase one. Couchbase handles better this situation when the machine is master for a part of data and slave for another part of the data, in particular for data redistribution.
shard X and secondary of shard Y, but primary of shard X and primary of shard Y) on the same machine. In 5.3, we will show that it can make sense.

### 3.3.2 Couchbase architecture

Sharding will be discussed first since unlike MongoDB it is necessary to shard in order to replicate data (inside a cluster, see 3.3.2.4 for extra-cluster replication).

#### 3.3.2.1 Sharding

**Architecture** Couchbase can distribute a bucket (document container) within different machines by putting a subset of data on each of them. As shown on figure 3.5, each machine contains a part of active data (replication will be studied in next section) and manages its own index.

![Couchbase architecture](image)

**Figure 3.5: Couchbase architecture**

Source: Couchbase whitepaper: Couchbase Server Under the Hood [44]

**Data splitting** As shown in figure 3.6 a document is identified by its key (id) and is contained in a document container called a bucket. Each bucket is split into 1024 vBuckets (logical partition, shards) by using a hash function (CRC32). Each vBucket is then distributed in different machines of the cluster [45, p 3]. Each vBucket is mapped to a machine, this mapping is stored in a look-up structure called

---

20] If we said that sharding enables to split data across several machines, it actually enables to split data in different folders which in particular enables to split data in several machines.

21] Hash function is configurable.
the cluster-map. This cluster-map is replicated to every client. In case of rebalance, the vBucket are moved by the balancer to the other node. If a machine is added the cluster map is modified to map to a machine more (it is also the case for node failure).

![Figure 3.6: Couchbase: Mapping of documents key to partitions and then to server](source: Couchbase whitepaper: vBuckets: The Core Enabling Mechanism for Couchbase Server Data Distribution [45])

**Query routing** When we retrieve a document using the `get()` function, the hash based on the document key is computed. This hash determines which logical partition (vBucket) the document should be retrieved from. Then using the cluster map, the database is able to determine in which physical machine the data should be retrieved from. Note that the routing process is done on the client side itself. The mechanism for insertion using the `set` function is the same.

However if the query is a search and therefore based on "views", the process is different (unlike MongoDB), queries on views are sent to a randomly selected server of the cluster [44, p 9]. The server which receives the query sends the request to other machines in the cluster, and merges the results from them before returning it to the client.

Note that a retrieve (`get()`) is by default consistent and a search based on a view is by default inconsistent (we might read stale data).
**Kind of shard key**  As seen in query routing section, Couchbase computes vBucket based on the hash of the key (hashed based sharding on the key). There is only this possibility in Couchbase. However the hashing function for vBucket mapping is configurable [45, p 3].

**Query isolation**  Query isolation is provided for `get()` and `set()` operations which can directly target the correct machine\(^{22}\).

### 3.3.2.2 Process and physical machine

Couchbase does not manage replication and sharding independently as we will see in the next section. We don’t have access to the process level.

### 3.3.2.3 Replication

**Read and write**  As shown in figure 3.5, each machine manages a subset of active (master) and replica partitions (vBucket). For a given vBucket only one copy is active on a machine with replicas of this data on other machines (the same vBucket can be replicated up to three times inside the same cluster). The active vBucket receives all read and write\(^{23}\) and replicates them to "replicas" vBucket. We can decide if replicas should build indexes or not [38, p 222]. By default clients read only from the active vBucket, however it is possible to allow clients to read from replicas but at the cost of reading stale data even for get operations. In Couchbase when a document is inserted the following happens (figure 3.7):

- Document is written to managed cache (RAM).
- Document is added to intra-cluster replication queue to be replicated to other machine(s).
- Document is added to disk queue to be persisted into memory.
- XDCR push replicates to other cluster(s) (this feature will be discussed later).

\(^{22}\)Unlike MongoDB, in Couchbase only get operations can make targeted query and not the search (views) one.

\(^{23}\)Note the process is different from `get()/set()` and views as seen in 3.3.2.1.
Failover In case of node failure, all vBuckets (partitions) which were active on that node will not be available. Couchbase will detect this and promote replicas of documents which were on this server to active. The cluster map will be updated to take into account this.

Write concern Usually Couchbase returns the hand when the data is written to the RAM where the bucket is active, but if we care about not losing data, we can wait for data to be replicated to disk. We can also control if the inserted data has been replicated to a specific number of replicas and also persisted into disk of these replicas. It is also possible to control the insert asynchronously.

Read isolation We saw that in MongoDB, a document was available for search as soon as it was inserted, thus it also updates indexes at insertion time. Using Couchbase (without XDCR cf.3.3.2.4), after a write, the cluster map is updated and the document is available for retrieve (get()) operations. However for search operations, Couchbase is using views and updating of these secondary indexes is done at the point of data querying, rather than each time data is inserted. This update at query time can be controlled with the stale parameter which can take the following values:

- stale=ok means: The index is not updated. The existing index is used to answer the query and return the results based on it. This gives fastest response
time as there is no need to update indexes but documents which were inserted after the last update might be not included in the response whereas they exist in the database [46].

- stale=false means: The index is updated before the query is executed. This ensures that any documents updated (and persisted to disk) are included in the view. The client will wait until the index is updated before running the query therefore response time will be impacted [46]. Note also that Couchbase needs to wait the replication to disk before indexing data.  

- stale=update_after (default) means: The existing index is used to answer the query, but the index is marked for updating after results have been returned to the client [46].

However views are actually built to run asynchronously, while doing our tests we noticed that when setting stale to false, the response time was particularly long and not acceptable. A strategy to make faster search could be to run the view with stale = update_after each insertion or every k seconds (k depending on how many documents it is acceptable not to return or update).

### 3.3.2.4 XDCR replication

Couchbase has a special feature called XDCR for cross data-center replication which is a master-master replication and enables to replicate data across multiple data-centers as shown in figure 3.8.

![Figure 3.8: Couchbase Cross Data Center Replication (XDCR)](image_url)

Source: Couchbase whitepaper: Extend your data tier with XDCR. [47]

---

27 Unlike MongoDB.
28 The replication we talked about previously was master-slave replication even if each server was master and slave at the same time, it was not what we call master-master replication.
29 XDCR is activated per bucket, some buckets can be replicated into another data-center and some can not.
Chapter 3. Document store

All we described previously in Couchbase is usually located in one data-center\(^{30}\). For some applications we might need two geographically diverse data-centers, for instance a data-center in America and a second data-center in Europe. This will enable [44, p 12]:

- To recover for disaster recovery: if an entire data-center is down our application will still continue to work.
- To bring data closest to the customer: a transatlantic round-trip costs a lot in terms of response time and high SLA might force having two remote data-centers.

XDCR might lead to write conflict as replication is bidirectional (active-active), for instance if the same document is modified in two remote data-centers. Couchbase manages write conflicts by keeping the document with the most modifications. If Couchbase guarantees strong document consistency within a cluster\(^{31}\), XDCR guarantees eventual consistency across cluster [47, p 4], it means that a document inserted in the US might not be accessible in Europe directly after insertion, but at some in the future it will be.

In order to avoid write conflicts we could:

- Accept and manage conflicts such as DynamoDB and Amazon shopping cart.
- Configure XDCR in unidirectional mode. In that case a data-center will be the master and another one the slave (not master-master replication anymore). In that way we can be sure the correct data is located on the master data-center. We will be able to perform local read using both data-centers (but read from the slave data-center will be eventually consistent). The major drawback of that method is that if a write is done from the US and the master data-center is in Europe, we will have to pay a transatlantic round-trip.
- Configure XDCR in master-master mode and route some property to one or another data-center. This will reduce write conflicts as booking for a specific property will be routed to a determined data-center. This will also enable to perform some local writes as some property might be more booked from one continent than another one.

\(^{30}\)But it can be in two and it would make no sense in Couchbase. Because machine holds active and replicas data and there is no tag aware sharding unlike MongoDB. See 3.3.3 to see how MongoDB uses this. Couchbase actually does not need this with its XDCR functionality.

\(^{31}\)For retrieve when not reading from replicas, not search based views are usually asynchronous but can be synchronous as seen previously.
3.3.3 How we can we bring data closest to customers without XDCR using MongoDB?

MongoDB does not allow master-master replication, and it might be a problem if it is necessary to bring data closest to the customers as Couchbase XDCR enables. In this section we study how MongoDB can solve the data multi-locality problem.

3.3.3.1 Use replica set in two different data-centers

![Replica set distributed in two data-centers](image)

Figure 3.9: Replicas set distributed in two data-centers

A shown in figure 3.9 we can use a primary and a secondary on one data-center (Europe) and another secondary\textsuperscript{32} in a remote data-center (US). For read operations the customers in Europe and US will be able to read from the nearest server, and thus avoid a transatlantic round trip. As a reminder since we read from secondary we accept to read stale data. However if we perform a write operation from the US, since only the master of a replica set can accept write (and this is because MongoDB does not support master-master replication), we will have to pay a transatlantic round trip.

This is equivalent in terms of functionalities to \textit{unidirectional} XDCR replication.

3.3.3.2 Use tag aware sharding

MongoDB supports tagging a range of shard key values to associate that range with a shard or group of shards. Those shards receive all inserts within the tagged range. Thus we can put data inserted from the US in the US data-center and the one from Europe in the European one, in that way we won’t have to pay a transatlantic round trip.

\textsuperscript{32}So as to avoid to make primary change continent, we should set priority of this secondary to 0 to prevent election as a master in case of node failure.
trip at insertion. However data will be in US or in Europe, except if we combine this technique with the previous one.

### 3.3.3.3 Combine the two

As shown in figure 3.10 we can combine the two techniques, in that way we are able to perform local reads and writes. The entire database content will also be in the two locations. However, assuming the entire EU data-center is down, as we always want the primary of a replica set containing bookings made in Europe in the European data-center, the US data-center will not be able to elect a new primary for this shard. Thus content of this shard will be only available in read only.\(^{33}\) If there is a network outage between the two data-centers, with the technique explained we should be able to continue to write and read data.\(^{34}\) This is almost equivalent to bidirectional XDCR. The advantage of Couchbase architecture is that there is no dependence between the two data-centers: if one is down they can continue to work independently unlike MongoDB. However, XDCR replication does not guarantee consistency in write, and might lead to write conflicts. Couchbase is actually said to be CP at the document level in a data-center and AP across several data-centers. Whereas when using MongoDB in that configuration, since write can be

---

\(^{33}\) Also in the configuration shown it won’t be able to elect a new primary due to “the majority of nodes in the set available in the replica set to elect a master” rule. To solve this problem we could however put an equal number of members for each replica set in the two data-centers, plus a replica in a third data-center [38, p 180]. This would provide always an available majority even if an entire data-center is down, but this would involve having a third data-center and also we would not be able to ensure that the data-center for data written in Europe would stay in Europe.

\(^{34}\) For the one coming from another data-center, only those replicated before the outage are obviously readable.
only done in the primary, even in a distributed data-center there won’t be any write conflict and consistency can be guaranteed at document level using journaling\textsuperscript{35} and write to the majority of replicas\textsuperscript{36}. In order to reduce write conflicts we could as mentioned previously route booking made by a specific property to an assigned data-center. Nevertheless both bidirectional XDCR and this MongoDB configuration will not guarantee consistency in read and lead to read stale data (MongoDB will read from secondary here). Also in the case of several shards (or even a third location), MongoDB architecture will become more complex than Couchbase one.

### 3.4 MongoDB and Couchbase mapping

In this section to understand Couchbase and MongoDB concepts, we will map MongoDB concepts to Couchbase ones. First a bucket is on the same level as a MongoDB collection or database. The collection concept does not exist in Couchbase, but it is possible to put a type field in documents within a bucket to distinguish two kinds of documents.

Figure \ref{fig:3.5} shows Couchbase architecture. To have an exact equivalent architecture with MongoDB, firstly in one data-center, we would shard the collection based on the hash of the id field. One machine would be master of part of data and slave of another part, therefore it would be something similar to the configuration seen in \ref{3.3.1.3}. Also Couchbase clients holds the cluster map, and direct the query to the good shard, whereas in Mongodb clients connect to MongoS which route the query to the good shard (good machines). Therefore to have an equivalence in terms of concept we should have as many MongoS as there is clients (this not recommended and just to explain the difference). Also we saw that Couchbase splits the data in 1024 vBuckets, this would be equivalent in MongoDB to run shards, where the sum of chunks in each shard is 1024\textsuperscript{37}. To get exactly what Couchbase (bidirectional) XDCR is, in MongoDB, master-master replication should be supported by MongoDB but it is not (at least officially \cite{48}). In that case a primary could replicate to another primary and we would get the kind of configuration shown in \ref{3.11}. The drawbacks of the lack of this functionality were explained in \ref{3.3.3}.

\textsuperscript{35}Journaling enabled and \texttt{j = True}.
\textsuperscript{36}\texttt{(w:majority)}.
\textsuperscript{37}We could also say that it would be equivalent to have 1024 shards.
3.5 Conclusion

3.5.1 Functionality

From a functional point of view, MongoDB has a better handling of documents than Couchbase. It offers powerful indexing options including text search and provides a real query language (described in 3.2.4.1) offering more possibilities than Couchbase and an aggregation framework. Also it separates query creation from index definition. Therefore for reservation search purposes MongoDB would fit better.

3.5.2 Architecture

We have identified that MongoDB architecture is more flexible than the Couchbase one (choice of the shard key, sharding and replication are separated which enables to build different architecture as shown in 3.3.1.3, if we want replication in Couchbase we are finally forced to shard). However the lack of master-master replication (XDCR) in MongoDB can be a showstopper if we need to perform low-latency reads and writes in two different locations with no interruption of service in write if one entire data-center is down (high availability).

3.5.2.1 CAP Theorem and BASE

In chapter 2 section 2.2 we studied CAP theorem and base properties. We said that DynamoDB was an AP system and depending on its configuration (R,W and N parameters) it could turn into a CP system.
Similarly MongoDB is a CP documents store, but by activating reading on replicas [49], this will make the system more available but less consistent in read. Same for write concern [50]. Thus if MongoDB is told to be by default consistent at the document level\(^{38}\), this consistency is lost when configuring MongoDB for availability, turning it into an AP system.

Couchbase is also considered as a CP document store. Actually inside a cluster, Couchbase is consistent at document level for \textit{get()/set()} operations\(^{39}\) and eventual consistent when querying views. Between two clusters when using XDCR, Couchbase is not consistent but eventually consistent. Thus Couchbase with XDCR is more an AP system than a CP one.

In DynamoDB a statistic approach was used to determine correct data, but this approach is not relevant in MongoDB and inside a Couchbase cluster (intra-cluster replication) since we know that in case of conflicts correct data is always stored in the master node.

Regarding BASE properties, we can notice that MongoDB and Couchbase are not exactly eventual-consistent and can actually offer some consistency level.

### 3.5.2.2 Synthesis

Our results are summarized in the figure 3.12 where MongoDB and Couchbase are compared according to several characteristics seen previously.

### 3.5.2.3 Recommendations

From the pre-study, we have determined that:

1. Couchbase is a good choice to be actually used as fast key-value store but its document oriented functionality is not mature yet. If one of the requirement is to be able to perform not only read but also write in two data-centers in different locations even if an entire data-center is down, Couchbase is the database to choose for its XDCR functionality, but we will have poorer document handling functionality.

\(^{38}\)Consistency at the document level means that operations atomicity can be guaranteed on a single document unlike a multi-documents operations. This is due to the fact Couchbase and MongoDB does not implement transactions.

\(^{39}\)However if we allow reading on replicas \textit{get} will be also eventual consistent
2. MongoDB is a good choice for manipulating documents. The query language is easy and enables to make complex queries and its architecture is very flexible. If XDCR functionality is not needed, then its handling of documents and search functionality would make MongoDB a better choice.

The ideal would be to have XDCR and great handling of documents, if MongoDB supports a kind of XDCR it would make it great. Also there is no better database than the other one, but one which fits better the requirements we have. This is the reason why it is important to study deeply what they offer and not only look at benchmarks. Indeed we often found benchmarks showing that a database performs better than the other one, this is really instructive but we should also ask the good questions: because it is a challenge to compare two databases which have a different architecture. For instance the majority of benchmarks only test `get()/set()` throughput and not particular functionalities such as the search. Sometimes an operation looks like the same but is actually different. For instance if we test insertion response time, MongoDB will update indexes at insertion whereas we saw that Couchbase by default will update them at query time. Hence we would measure something different.
As one of our main goals was to improve search functionality and due to the fact MongoDB better supports document handling, we select MongoDB for our prototype.
Chapter 4

MongoDB prototype

To prove functional feasibility of using MongoDB for storing hotel reservations, we realized a proof of concept (POC).

4.1 Data model

First we need to define a data model.

4.1.1 NoSQL data modelling

In the relational world we take information to store and split it into rows (tuples). We cannot nest a row into another row whereas document databases follow the aggregation approach. An aggregate is a JSON document which contains data we want to treat as a unit [7, p 14]. The aggregation approach makes it easy to operate on data that have a more complex type than rows, typically it makes it possible to nest data. In relational databases, data normalization is a process often used to avoid data redundancy. Whereas when designing a NoSQL database, we often actually practice data denormalization. Another difference with the relational world is that a query can infer the data modeling itself. Finally MongoDB does not need to define a schema and does not have an instruction such as Create table. For instance let’s suppose we want to model that a customer has booked between two dates in a hotel which is identified by a property code and that the hotel belongs to a brand identified by a brand code. In a relational database, a solution would be to model
this using three tables: a customers table\(^1\), a bookings table\(^2\) and an hotels table\(^3\) such as shown in figures 4.1, 4.2 and 4.3.

<table>
<thead>
<tr>
<th>last name</th>
<th>first name</th>
<th>bookingID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coulombel</td>
<td>Sylvain</td>
<td>1</td>
</tr>
<tr>
<td>Coulombel</td>
<td>Emilie</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: A customers table

<table>
<thead>
<tr>
<th>bookingID</th>
<th>check in date</th>
<th>check out date</th>
<th>HotelID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20140801</td>
<td>20140810</td>
<td>LON212</td>
</tr>
</tbody>
</table>

Table 4.2: A bookings table

<table>
<thead>
<tr>
<th>HotelID</th>
<th>BrandID</th>
</tr>
</thead>
<tbody>
<tr>
<td>LON212</td>
<td>HL</td>
</tr>
</tbody>
</table>

Table 4.3: An hotels table

In MongoDB, one of the possible data model would be to put everything in a single document such as:

```json
{  
guest_names : [    
{last_name : "coulombel", first_name : "Sylvain"},
{last_name : "coulombel", first_name : "Emilie"}
  
  
  room_stay: {    
  
  check_in : 'date1'
  
  check_out : 'date2'
  
  hotel_info:{    
  
  property_code : "LON212",
  
  chain_code : "HL"
  
  }
  
  }
  
  }
  

```

In that way everything is contained in the same document which makes MongoDB very powerful when we need to make a search on one or combination of different fields and return aggregates matching the query\(^4\). A RDBMS would have to get information from multiple tables. However with the aggregate approach we duplicate data, since for instance each booking will contain that the hotel "LON212" belongs

\(^1\) containing customers information and a bookingID, a foreign key identifying the booking made ny this customer in the bookings table.

\(^2\) containing bookings information and a hotelID, foreign key identifying the hotel in which the booking was in the hotels table.

\(^3\) containing hotels information and the brand code this hotel is belonging to.

\(^4\) We could also project some fields of the aggregate.
to the chain identified by "HL". It also solves the impedance mismatch problem talked in 2.1.3.

### 4.1.2 Our modeling

The current RDBMS is actually already modeled as a document/key-value store. There is a table which contains a blob in its column. This blob contains a structured document (protocol buffer) called the e-voucher image containing booking information which is actually already an aggregate. Other columns of the table have been introduced to be able to find this blob. For instance the column name enables to search by name, when the row is found we extract the blob. To transform the current data model to one compatible with MongoDB we just need to transform blobs (protocol buffer documents) to JSON (BSON) documents which will be our value and index necessary fields for the search. A booking example is given in appendix A.

We have chosen to model the problem by putting all information regarding one booking in a single document for two reasons:

- MongoDB can guarantee consistency at document level, but not multi-document consistency (transactions are not supported, so if one booking is represented by two documents, nothing will guarantee that the two documents were inserted).
- MongoDB does not support joins.

The document (value) contains the following information/fields:

- **IDs**: Confirmation number, cancel number, and PNR record locator.
- **Transaction data**: Type (sell, modify, cancel) and the transaction date.
- **Property IDs**: Property code (hotel unique id) and chain-code (chain unique id).
- **Originator**: Identify who made the booking, typically a travel agency or website such as booking.com
- **Amount**: Information regarding the price and the currency.
- **Customers**: Information about customers such as name and address.
- **Room-stay**: Occupants, services, rate plans, stay-dates.
• Form of payment: Credit card information or other.

• Comments: Free text comments and language code in which comments were made.

We decided to put all bookings in the same collection, thus we have a field which identifies to which property a booking belongs. Another option could have been to create a collection per property, but this would have lead more maintenance problems, and would create too much collections in a cluster. However if the number of properties is very high, and as brand gathers a set of properties, a good option could be to create a collection per brand code.

4.2 Choice of a Framework and API

Choosing a programming language for a prototype is not an easy thing. MongoDB offers an API for almost all languages (C++, Java, Python, PHP...). We have chosen Python as it is a language simple of use and which enables to do fast prototyping. Also Pymongo, the Python API for MongoDB seems to be well supported. The goal of the POC is to be able to test MongoDB, but also provide a realistic environment, this is the reason why we decided to create a REST API which will request our database. The result should be returned as an HTML page or a JSON file. The REST API has been done with Tornado (Facebook’s web framework). It was chosen over Bottle or Flask because it enables to create nice web pages easily (which could be an evolution of the prototype, since it is not the goal of the project).

4.3 Supported operations by the prototype

A request is made to the database in the following way: http://server:8000/Operation?param_1="param_1"&param_2="param_2"&...&param_n="param_n" using the REST/JSON API. An operation is the kind of action we want to do (search by id, name, date...). Parameters are operation arguments, some are specific to an operation whereas other one are common to all operations. Commons parameters are:

• property_code: id identifying a property [Mandatory\textsuperscript{5}].

\textsuperscript{5}possible to omit but in that case it is recommended to change the index strategy or accept slow response time.
• state: The reservation state, which can be a sell, modify or cancel [Optional].

• displayType: HTML to display an HTML web page with the results, JSON to return a JSON array with all matching documents [Optional]

Here are described all operations available in the prototype with their specific parameters.

4.3.1 Search by ID : /searchByID

Search by IDs enable to make a search by confirmation number, cancel number or passenger number record. All these fields are unique. This is a functional retrieve.

Parameters specific to that query are:

• CF_ID (confirmation ID) or CX_ID (Cancel ID) or PNR Record locator [One of them is mandatory].

For instance an API request to retrieve a booking by id could be: `http://server:8000/searchByIDs?CF_ID=WRN4E8G90C&property_code=BLTLSP03&displayType=HTML`.

Behind the JSON document representing the query will be:

```
1 {  "proptyIDs.propty_code": "BLTLSP03",  "IDs.CF_NUM": "WRN4E8G90C" }
```

This will return the first two hundred documents in a HTML table.

4.3.2 Search by check in date and check out date:/searchByRangeDate

It makes it possible to search all bookings which check-in date or check-out date or both are (is) equal to a specific date are in a range of dates. Parameters are:

• Check-in date: customer arrives in the hotel at ci_start_date or between ci_start_date and ci_end_date

• Check-out date: customer arrives in the hotel at co_start_date or between co_start_date and co_end_date [one of the two is mandatory]
For instance the request: `http://server:8000/rangeDate?ci_start_date=20140715&ci_end_date=20140731&co_start_date=20140731&co_end_date=20140802&property_code=BLTLSP03&displayType=HTML` will return all the bookings between two ranges of date. The JSON document representing the query will be:

```json
{
    "room_stay.dates.end": {
        "$lte": datetime.datetime(2014, 8, 2, 0, 0),
        "$gte": datetime.datetime(2014, 7, 31, 0, 0)
    },
    "proptyIDs.propty_code": "BLTLSP03",
    "room_stay.dates.begin": {
        "$lte": datetime.datetime(2014, 7, 31, 0, 0),
        "$gte": datetime.datetime(2014, 7, 15, 0, 0)
    }
}
```

Also there is special parameter such as leavingToday (all customers leaving today), arrivingToday (all customers arriving today) and in house (all customers in the hotel): `http://172.16.225.157:8000/rangeDate?shortcut=leavingToday&displayType=HTML`

### 4.3.3 Search by name: /searchByName

It makes it possible to search booking by customers’ last name and first name. It takes as a parameter:

- `first_name`: occupants’ first name [Optional]
- `last_name`: occupants’ last name [Mandatory]

For instance the following request: `http://server:8000/searchByNames?last_name=coulombel&property_code=BLNCYP01&displayType=HTML`, will output all the bookings in which one of the last names of the guest is coulombel. Under the hood the query made to the database will be:

```json
{
    "customers.last_name": "coulombel",
    "proptyIDs.propty_code": "BLNCYP01"
}
```

Note that we can change this query to find all customers\(^6\) which last name starts with "coul", for instance that way:

\(^6\)We used exact match in our benchmark but it could be interesting to study if search by start with performs as well as an exact match.
In chapter 6 more details are given on search in a array. Also search is case sensitive, that's why when inserting and doing search we normalize documents and removed capital letters.

### 4.3.4 Search by transaction date

It makes it possible to search booking by transaction date. The API parameter is:

- `transaction_date`: the date when operation was made

For instance the following request `http://server:8000/searchByTransactionDate?start=20140418_130000&end=20140418_140000&state=initiate&property_code=BLNCYP01&displayType=HTML`, will output all the bookings which were made between 1 and 2 PM on the 18th of april 2014. The query document will be similar to a search by check in or check out date.

### 4.3.5 Search by multi: /searchByMulti

It makes it possible to search a booking by mixing all parameters belonging to the query above. For instance the request `http://172.16.225.157:8000/searchByMulti?ci_start_date=20120101&ci_end_date=20121201&last_name=harris&state=SELL&property_code=BLNCYP01&displayType=HTML` will return all bookings which have a check-in date between the range of date and where customers is equal to Harris.

### 4.4 Implementation

As some fields can be left blank in the query, either we can make the database test if a field is not null but this introduces uselessness checks on server side. We used a different approach, as queries are JSON documents (dictionaries in Python), the front-end generates a dictionary which represents the query dynamically, in case a field is null we do not append it to the query document.
4.5 Creating good indexes

4.5.1 Theory

An index is also created using a JSON document in native API. In Python API it is an array. The document or array contains key(s) and a value(s). The key is the field to index and the value is either the sort order (1 (pymongo.ASCENDING) for ascending, -1 for descending (pymongo.DESCENDING)), but can be also used to refer to a special index such as "GEO2D" or "text". MongoDB supports 5 kinds of indexes which are single field, compound (index several fields), multikey (index an array), geospatial, text (full-text search) and hash indexes described in appendix B.

Compound index is a particular case where the index contains several fields of a document. For instance if we are making a search on customer’s name by hotel (property code), we will index the hotel code and the name field. It is even possible to make a compound index with an array, full text or geospatial field. For instance when using text index, we could create a compound index with a prefix of a criterion different than the text search such as the property code followed by the full-text fields. In that way we would narrow the search by another criterion and would make the query faster [38, p 119]. However there are some limitations, if we create a compound index with a geospatial field, at the moment of writing this field must be the first one in the compound index.

When no sort parameter is specified MongoDB will return the result in index order (but it is not guarantee). If we query and sort on a single field in descending order whereas a single index on this field has been created in ascending order, MongoDB will use this index efficiently as it can read indexes in the two orders. If the sort is done on several fields and compound index is used then the order in the query and index will have its importance, this will be studied in the search optimizations part in chapter 5. In the test scenario we didn’t add a sort option, however for usual sort this is done at no cost as MongoDB will already read the index in the good order.

An index on the _id field is always created by default in each MongoDB collection.

---

7Same for sort parameters.
8In the documentation it is said “A compound index includes more than one field of the document”, we prefer to say index instead of field because a text indexes can actually index one or several fields, an array containing two integers can be indexed as a multi-key or a geospatial index.
9For instance if we search all bookings which check-in and check-out are between a range of date for a specific hotel and a compound index on property code, check-in and check-out date is used: result will be sorted by check-in and check-out date for free.
4.5.2 Strategy

For each operation except the "searchByMulti", we create compound indexes containing all mandatory keys with the property code as a first key. Note this property code is used by our of all queries. In chapter 5, we will verify if this was the best strategy. For instance we recommend to create a compound index on [property code, check in, date, check out date]. If this index is able to be used on queries which contain only the "property code" or the "property code and check in date" or "property code and a check-in date and an check-out date". Queries containing "property code and a check-out date" are not fully supported by this index (it can only use the property code part). Therefore we also advice to create an index on [property code, check-out]. Similarly an index on transaction-date, first-name and last-name should be created. We can also create an index for functional retrieval on confirmation number, cancel number and PNR.

The proposed compound index does not include the non mandatory parameter\textsuperscript{10} "status" because this would make grow the index for a small percentage of our queries which includes the status\textsuperscript{11}. Moreover as this field has a low cardinality it is not very helpful.

It has been noticed that index size increases linearly with the number of bookings in the collection. When creating a compound index with several fields it is important to design documents with short field names, that’s why we reduced some field names such as "property" to "propty" or "Transaction" to T. Otherwise this might lead to the error: " ns name too long, max size is 128", which is better anticipate as it might be a pain to change this later.

Regarding the search on any fields (not included in test scenario) we did not create a specific index for this query as it would imply a lot of combinations thus a lot of indexes. MongoDB has also an index intersection feature but with our data and existing compound index, the query engine will most likely never use it for this request. One of the reasons is that the use of one of the compound indexes might be more selective and scan less documents than trying to intersect two indexes. However an approach could be to take advantage of index intersection to create less compound indexes but index intersection will not perform as well as a compound indexes.

\textsuperscript{10}Optional parameters such as status should be added at the end of the index.
\textsuperscript{11}None will be in benchmarks.
4.6 Conclusion

For the search of hotel reservation use-case, MongoDB enables to perform the needed functionalities (search by range of date, name, start with name ...). In chapter 5, advanced search functionalities will be described. We will now focus on performance tests which also enable to give configuration recommendations.
Chapter 5

Performance tests

A simple booking system to search reservations using MongoDB has been realized. However benchmarking the needed functionalities is necessary to validate the feasibility of using MongoDB. The focus will be put on response time, throughput and scalability.

5.1 Testing framework and configuration used

5.1.1 Testing framework

In order to perform tests, a testing framework has been written in Python. A testing framework is necessary to fill the database and generate queries on this database according to hotel reservation scenarios for different configurations.

5.1.1.1 Populate the database

To populate the database, there are two options, we can import real data or generate random data. We choose to generate random data to have more freedom, and since it enables to fill the database with more inputs than there is in the current one. However the drawback of generating random data is that the randomness should respect the distribution there is in the real database. For instance this is particularly true for search by name functionality where names frequencies might impact search performances.
For this purpose a booking generator class has been written. It generates random bookings which are stored in the database. To fill the database faster, in development phase, we can use bulk insert which enables to insert an array of documents in the database. Since it is needed to respect the same name distribution as there is with real data, a list of names of the current Oracle database with their frequencies has been extracted. It enables to generate random name following the same distribution as there is in reality. Another parameter which can have impact on search performance are stay dates (duration between the check-in date and check-out date). The average stay duration is about three days. Therefore random generated booking will have a room-stay which is between one day and four days (this parameter can be modified later in a configuration file). Now that we have data in the database, we will need to query this database.

5.1.1.2 Query generator

**Methodology**  A query generator class has been written and generates random queries on the system according to a given distribution. In MongoDB, when using Pymongo driver, queries are represented by a Python dictionary (JSON file in native JavaScript API), therefore the query generator generates a Python dictionary (we did not query through the REST/API directly). The query generator can generate queries of search by check-in date, check-out date, transaction date, first name, family name and property code and the status\(^1\). This query parameter can overlap therefore we can for example make a query combining check-in date, check-out date and property code. The search parameter distribution can be modified in a scenario file.

**Workload in Search definition**  For our test we have chosen a distribution close to real ones. In our test all queries contain the property code and another search criterion with this distribution :

- Property code and range of check-out date: 30%.
- Property code and range of check-in date: 20%.
- Property code and range of check-in date and check-out date: 10%.
- Property code and range of transaction date: 15%\(^2\).

\(^1\)Status was not used in our test scenario.

\(^2\)In the test scenario booking’s transaction date granularity was day, we realized it would have been better to take a ms granularity. However this should not affect results.
• Property code and search by family name\(^3\): 20%.

• Property code and first name: 5\(^4\).  

Note the response time and throughput are mainly dependent on the workload.

**Range date distribution** For reservation search by check-in or check-out date, we specify the query as a range of date, between date A and date B. if the difference between the two dates is high, the query might take more time to run. For our test we considered that the minimum difference between these two dates is 2 and that the maximum is 30. When we search by a range of check-in and check-out date at the same time we choose that the distance between the check-in start date and search check-out start date would be between 1 and 3 in the search request. This can also be modified in the configuration file to try different configurations. When querying the database we then only returned the first hundred results. Also in the documents stored in the database room-stay length (number of days between check-in and check-out) were between 1 and 4 days and can also be modified.

### 5.1.1.3 Indexing

Here are the indexes which have been created\(^5\) on the collection adapted to our query set for performance tests:

```
1 { "_id" : 1 } // created by default
2 { "proptyIDs.propty_code" : 1, "room_stay.dates.begin" : 1, "room_stay.dates.end" : 1 }
3 { "proptyIDs.propty_code" : 1, "room_stay.dates.end" : 1 },
4 { "proptyIDs.propty_code" : 1, "T_data.T_date" : 1 },
5 { "proptyIDs.propty_code" : 1, "customers.first_name" : 1 },
6 { "proptyIDs.propty_code" : 1, "customers.last_name" : 1 }
```

\(^3\)We made an exact match and did not do a start with, it would be interesting to study more deeply performance with a start with. Also to reflect better the reality around 20 percent of search by exact name returns an empty results. Note a search by name usually returns around 20 results, whereas search by date returned around 100 results when limit was set to 100.

\(^4\)difference with the family name is a lower variety of names.

\(^5\)In reality before sharding on the property code, we index property code on descending order such as { "proptyIDs.propty_code" : -1 }, this does not impact performance result. However as MongoDB return result in index order, this might have returned result in a different order. This was necessary because sharding on property code requires ascending keys.
5.1.2 Search and Insert

For testing search and insert performance, we wrote a Python script which used the booking and query generator. They follow the following algorithm:

\[
\text{for } i \leftarrow 1 \text{ to } 5000 \text{ do } \\
\qquad q = \text{GenerateRandomQuery()} // \text{We generate a query using the query generator} \\
\qquad \text{executionTime} = \text{run}(q) // \text{We run the query, it return the execution time} \\
\qquad \text{log}(\text{executionTime}) // \text{We log the execution time in a file} \\
\text{end}
\]

Algorithm 1: Search algorithm

The insert script is similar except that we generate a random booking and insert it into the database. We had also another script to insert bookings to fill the database.

Note that each script logs the time in a file. Another script measures the number of TPS (Transactions (operations) Per Second) from the database. For this test we use one (powerful) machine which hits the database with search or insert operations. As one of the goals is to determine how MongoDB will behave under heavy load (in search or insert), in order to increase the throughput, Python scripts were launched several times using a shell script. At iteration \( k \) of the shell script, we launch \( k \) searches (or \( k \) inserts) Python scripts\(^6\). As output a folder named \( k \) contains a log file from the TPS script which enables to compute the number of TPS reached by the database during the test, and each of the \( k \) process outputs a file, which contains execution times for each query they made to the database. By merging these \( k \) files it enables to make statistics and graphs on response times for a given number of TPS. We used the R language for this purpose.

5.1.3 The different Sharding configurations

5.1.3.1 Increase shard number

We perform test in a non sharded\(^7\) and a sharded environment with three shards and six shards. Here are the details on how the cluster was configured, where BMXXXXX is a machine name:

- Sharding disabled configuration: One primary shard on BMX10100

---

\(^6\)Firstly we used the Python thread library and realized there was a problem due to GIL (Global Interpreter Lock), therefore it was better to launch several Python scripts from a Shell script. We could also use the Python multiprocessing library (we used it for the workload generator).

\(^7\)At that time we already used MongoS to connect to database, but could have used MongoDB directly.
• Three shards configuration: Three shards on BMX10100.

• Six shards configuration: Three shards on BMX10100 and three shards on BMX10200.

### 5.1.3.2 Different shard keys strategies

The shard key determines the distribution of a document in a collection. The shard key is a single index field or a compound index. We try two different shard keys.

**Shard on hash of the id**  When we shard on the hash-id, it means we shard on the hash of the document key, for this we need to add another index of hash type, marked the collection (tesDB) to enable sharding and shard on that key as the following in a Mongo shell connected to MongoS:

```javascript
1 db.bookingCollection.ensureIndex({_id: "hashed"})
2 sh.enableSharding("tesDB")
3 sh.shardCollection("tesDB.bookingCollection",{ "_id":"hashed"})
```

**Shard on hash of the property code**  When we shard on the property code it means we shard on property code, check-in date and check-out date. So we used the existing index on that field, also enable sharding on the collection as previously and shard on that key as the following:

```javascript
1 sh.shardCollection("tesDB.bookingCollection",{ "proptyIDs.propty_code": 1, "room_stay.dates.begin":1, "room_stay.dates.end":1})
```

**Justification**  We believe a compound shard key on a the property code, check-in and check-out date (range based sharding) seems to be a good choice because:

• Query on property code, range of check-in and eventually check-out date are the most common one and range based sharding works well for that kind of query.

• All queries contains the property code, therefore MongoS will be able able to perform query isolation as discussed in 3.3.1.2.

• Having the check-in and check-out in the shard key and not only the property code will enable chunks to split better as seen in 3.3.1.2.
• As seen in 3.3.1.2, the drawback of range based partitioning is that it can result in unbalanced clusters. This is particularly true when sharding on ascending keys only. For instance if we shard only on check-in and the check-out date without the property code, all write operations will be routed to the same shard by the MongoS (as this dates increases with the time), and the balancer will have to migrate data to different shards to keep a well balanced cluster. Thus this will prevent the database to scale well in write. However if we add the property code before the date keys, it will result in an almost ascending key but for each property. This will distribute well the data across the cluster. Indeed insert operations will now not always target different shards. Hence write operations will be well distributed and search operations will be able to benefit of the query isolation mechanism.

The hash based partitioning might also be a good choice as it provides an even distribution (most likely better than the compound shard key) of read and write as seen in section 3.3.1.2, but we suspect that the query isolation process on the property code will be a great advantage given that our query always contains the property code, and that the proposed shard key might scale better.

5.1.4 Client and MongoS configuration

The Python client will be running on BMX10201 machine. The same machine will also host the MongoS process.

5.1.5 Hardware and software configuration

5.1.5.1 Hardware configuration

**Machine** In the table 5.1 we give the configuration of the physical machine\(^8\) used and how they were used. All microprocessors were Intel(R) Xeon(R) CPU, in the table we specify the model used for each machine and their number of cores.

Machines are also equipped with FusionIO cards. These machines are located in Erding (Germany) in the Amadeus Data center. During our first tests we launched scripts from Nice from several clients (not as powerful as BMX10201) using SSH loop, however we realized the network was the bottleneck. This the reason why we also moved clients to Amadeus data center.

\(^8\)Not virtual machine.
Table 5.1: Physical machine configuration

<table>
<thead>
<tr>
<th>Machine name</th>
<th>RAM</th>
<th>CPU model</th>
<th>Core</th>
<th>1 shard</th>
<th>3 shards</th>
<th>6 shards</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMX10201</td>
<td>192 GB</td>
<td>X5660 @ 2.80GHz</td>
<td>12</td>
<td>Clients and MongoDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMX10000</td>
<td>512 GB</td>
<td>X7542 @ 2.67GHz</td>
<td>24</td>
<td>Will be used later</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMX10100</td>
<td>256 GB</td>
<td>X7550 @ 2.00GHz</td>
<td>24</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>BMX10200</td>
<td>192 GB</td>
<td>X5660 @ 2.80GHz</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Network latency between the different machines

<table>
<thead>
<tr>
<th>From/To</th>
<th>To BMX10000</th>
<th>To BMX10201</th>
<th>To BMX10100</th>
<th>To BMX10200</th>
</tr>
</thead>
<tbody>
<tr>
<td>From BMX10000</td>
<td>0.017</td>
<td>0.069</td>
<td>0.085</td>
<td>0.105</td>
</tr>
<tr>
<td>From BMX10201</td>
<td>0.067</td>
<td>0.009</td>
<td>0.054</td>
<td>0.100</td>
</tr>
</tbody>
</table>

5.1.5.2 Software configuration

Tests were run on MongoDB 2.6 and using the Pymongo driver. All the dataset entirely fits in RAM, in that way it avoids page faults. A page fault is a situation to avoid when the data is not in memory and therefore the database needs to hit
the disk and hence response time is slower (in our case with FusionIO disk the impact would have been lower than if we were using spinning disk). The MongoDB command for preheating\(^9\) the data has been used. This loads all documents and indexes of a collection in memory:

\begin{verbatim}
1 db.runCommand( { “touch” : “tesDB”, “data” : “true”, “index” : “true” } )
\end{verbatim}

For insertion test we used a default write concern (journaling enabled on database and on client side a write level of 1 and journaling to false (w=1 and j=False))\(^10\). For insert test, the database was empty and we perform insertion. For search test the number of bookings (documents) was around 30 millions.

5.2 Sharding not activated: Throughput and response time test

In this section we introduce our throughput and response time results for search and insert. The goal of this test is to determine the response times and the maximum throughput MongoDB is able to reach for our configuration and use-case.

5.2.1 Insertion time

5.2.1.1 Hypothesis

This test enables to determine the impact of indexes on insertion time. Indeed if indexing enables to make search faster, we might pay it in insertion time. We expect a reduction of insertion performance when indexes are created which we will try to determine.

5.2.1.2 Results

Figure 5.2 shows insertion time according to the throughout and 5.3 shows the throughput according to the number of clients (processes).

\(^9\)Moving database into RAM.

\(^{10}\)If MongoDB is used as a master database, it will be worth to try a configuration with several replicas and journaling enabled on the database side, and then set j = True (indeed as seen previously j=False does not guarantee consistency) and try for different write concern levels (w = 1 to the number of members in the replica set) on the client side.
5.2.1.3 Interpretation

As shown in figure 5.3 when no indexes are created, the maximum insertion throughput MongoDB is able to reach is a factor of 11000 ops/sec. However if indexes are created, the maximum throughput it can reach is a factor of 3500 ops/sec. Therefore indexing reduces the maximum throughput the database can reach by three. In figure 5.2, it can also be seen that insertion time is also higher when indexes are created. If index enables better search performances, these experiments show the impact of indexing on insertion time. The more index there is, the slower the
insertion time is. That’s why we should be careful where adding indexes and not ignore this has a cost.

5.2.2 Search performances

5.2.2.1 Hypothesis

We now do similar operations for search but only when there are indexes. Indeed we run tests without indexes and search performances where not acceptable at all. The goal of this test is to measure the search performances we can reach with MongoDB. We expect a low response time until a threshold value.
5.2.2.2 Results

Figure 5.4 shows search response times according to the throughput and 5.5 is the throughput according to number of clients (processes).

5.2.2.3 Interpretation

The response times in search are very satisfying as shown in figure 5.4, the maximum throughput reaches seems to be 2000 TPS, however during our test we noticed that CPU consumption (top unix command) was very high. Therefore we strongly suspect the client to be the bottleneck, unlike at insertion time where the database is probably the bottleneck11.

5.3 Scalability tests

In this section we study how sharding impacts results.

5.3.1 Insertion

5.3.1.1 Hypothesis

During our first experience with sharding disabled it has been noticed, that insertion throughput was limited due to locks at the MongoDB level. Since we have good machines our hypothesis is that by adding more MongoDB and thus more shards, even on the same physical node could improve insertion performances, in particular throughput since the lock will be distributed among several MongoDB. This test will enable to check if this is making sense although it does not sound natural using the 3 shards configuration. Next we will use the 6 shards configuration where two machines will be used. We expect then a better throughput than with the three shards configuration. For that purpose we will compare with the previous results where the collection was not sharded (one shard configuration) with the three shards and six shards configuration, with hash-id as shard key and indexes built.
5.3.1.2 Results

Figure 5.6 shows insertion response times according to the throughput when sharding was disabled, and when the cluster was sharded on hash-id with the 3 and 6 shards configuration. Figure 5.5 is the throughput according to number of clients (process) in the same conditions.

At that moment of the work, we did not have a second client machine at our disposal, but it would have been interesting to launch search/insert from another client to verify if we could reach an upper TPS limit, this is what we do with the sharding configuration and BMX10000 machine as a client.
Chapter 5. *Performance tests*  

5.3.1.3 Interpretation

Our hypotheses have been verified. Sharding increases insert throughput even on a single machine.

5.3.2 Search

5.3.2.1 Hypothesis

Our hypothesis is that sharding should increase the throughput but should not necessarily improve search response time. For this purpose we will run search tests for the two sharded configuration (3 shards and 6 shards) when the shard key is the property code and hash-id, and we will compare the results with the non sharded configuration.

5.3.2.2 Results

**Shard key was hash-id** Figure 5.8 is search response time according to throughput with sharding disabled and when the cluster was sharded on hash-id with the three and six shards configuration.
Figure 5.9: Search response time with sharding disabled and when the cluster was sharded on the property code with the three and six shards configuration

**Shard key was property code** Similarly figure 5.9 represents the same experiment except that the shard key was the property code.

### 5.3.2.3 Interpretation

In both cases it is easy to observe that search response times increase. This is explained due to the fact that when we shard we actually had more complexity, a process (MongoS router) needs to determine the good shard(s) where to send the query and this has a cost. However the throughput seems to have decreased, this is a priori strange since in theory adding more shards should increase the maximum throughput. Our hypothesis is that the database is not the bottleneck and that the client machine is the bottleneck. Indeed if the client machine is the bottleneck, at the threshold point (even by adding more process to increase the number of TPS) the client won’t be able to generate as many TPS as it did before, because the response time increases due to sharding. Thus if the response time increases due to sharding, the throughput decreases due to client machine limitation.

The client machine being the bottleneck hypothesis will be verified in 5.4.

Also it sounds like the **median** response time is better when we shard on hash-id than when we shard on property code for low throughput value, whereas it seems to be the contrary for high throughput values. However the property code shard key seems to scale better as the 6 shards configuration response times are lower than
the three shards configuration, whereas when we shard on hash-id the contrary happens. This can be explained by the query isolation process seen in 3.3.1.2, indeed when we shard on property code, the isolation process enables to target directly the good shard(s) containing the interesting documents, therefore adding another machine is helping the process since the load is separated between two machines. On the other hand when we used hash-id as a shard key, the query router needs to contact all machines (scatter-gather), therefore adding 3 shards complicates the process as the query router will need to contact three more shards. The choice of a good shard key is very important and these experiments are not sufficient enough to take a decision, in section 5.6 we will study the impact of shard key on response time more deeply.

5.3.2.4 Comments about RAM size

Sharding is usually done to increase maximum reachable throughput or available RAM in the cluster. In our situation the entire data set always fits in RAM therefore we never had page faults. There are some situations, where one node would not have sufficient enough RAM to hold all of the data into it. In that case adding a second machine and sharding will increase the total available RAM in the cluster and will enable to put more data in RAM and thus reduce the number of page faults. In that case it could lead to a reduction of the response time. This might be even true if were using spinning disk rather than SSD\textsuperscript{12} disk.

5.4 Maximum throughput test in a sharded environment

We check if the client was effectively the bottleneck on the 6 shards configuration with the two different shard keys. For that purpose a script launching several processes (Python multiprocess) doing search (without logs to file) has been launched simultaneously from two machines (BMX10000 and BMX10200), the maximum throughput we could get when the property code was the shard key was 4000 TPS. When the shard key was on the hash-id, we were able to reach 1500 TPS. If we compare with figures 5.8 and 5.9 for the 6 shards configuration, we realized that a higher throughput was achieved. Therefore we have shown that the client was actually the bottleneck and it confirms the hypothesis stated below.

But the maximum throughput reached with the two different shards is huge (4000 TPS for the property code as a shard key and 1500 TPS for hash-id as a shard

\textsuperscript{12}Solid State Drive.
key). Actually to increase the number of TPS we need to increase the number of clients and thus the number of open connections. The problem when sharding on hash-id, is that since queries are all scatter-gather, MongoS needs to open even more connections to MongoD to answer the query unlike targeted queries when the shard key is the property code. As the number of connections a MongoS or MongoD can handle is limited, MongoDB returns this error when we tried to open more connections (clients) to reach more TPS:

```
1 err: couldn't connect to server bmx10100:27316 (172.17.208.37), connection attempt failed.
```

The number of maximum connections could be changed, therefore we could increase a little bit the number of TPS, but there will be a point where the operating system’s own limit will be reached.

This "error" also lets us think that the sharding on hash-id might not be the best solution for our use case, and that the property code as shard key seems to suit better.

Also when we reach a throughput of 4000 TPS, another machine running client would have been necessary to determine if we reach the database bottleneck or not. However 4000 TPS is sufficient for our use-case.

### 5.5 Criticism of test protocol and modification

Before studying more deeply sharding configurations, we would like to propose some details we could improve in the test protocol. Firstly we will identify some weaknesses of the previous protocol, and then propose a new test protocol correcting them.

#### 5.5.1 Problem identified

##### 5.5.1.1 Workload and measure are dependent

Clients (processes) used to generate traffic are the same as the ones we use to measure response time and throughput. This can lead to problem. Indeed if we want to measure search response times alone under a mixed workload of search and write traffic, it won’t be possible with the current test protocol. Another example is if we want to measure search by check-in date response time only under a workload of all kinds of query (check-in, name, transaction date ...). In the methodology used
as the clients used to generate workload are the same as the one used to measure performance we will only be able to have a workload of search by check-in date if we are benchmarking the search by check-in date. Whereas we would want to measure performance of this query under a workload of search of all kinds and even with some insert traffic. We could solve this problem by using a workload generator independent from the process which measures the response time. The workload generator would generate a workload using several processes distributed on one or several machine and would generate a desired traffic. Then a single process would run on another client and we would take measure from this process. However the drawback of this protocol is that we can not guarantee that everything is happening well on all clients (for instance a client could give an answer to a query very fast and another very slow, but we noticed it was not the case).

5.5.1.2 Throughput and response time are dependent on time

In the first test protocol, we measure response time and throughput from the database. Therefore throughput and response time are both dependent on the time. Hence graphs are in reality parametric graphs depending on t. This for instance explains why on figure 5.6 some points are turning back. The approach we propose to solve this problem is to write a program which generates a constant number of TPS, in that way we can control directly this parameter.

5.5.1.3 Use of workload generator and measuring process on same machine

Use of workload generator and measuring process on same machine make it confusing since the bottleneck is the client, we can’t establish for high throughput value if the higher response time is due to client or server or both in particular for high throughput value. A solution could be to use distinct machines for the workload and the measuring process.
5.5.2 Solution proposed

From now we will use a workload generator on BMX10000 which will generate a constant workload. The program generating the workload takes as input the number of TPS we want to reach$^{13}$. The workload generated can be a full search workload or a mix workload of search (of different kinds or not) and inserts. The search workload queries are those given by the query generator. Thus they respect the distribution given in 5.1.1.2. Then on the BMX10201 machine, a single process (client) is running and processing the operation we want to measure. Thus we are able to measure operation response time under a defined throughput.

The new configuration is now the one shown in 5.10 and the next performance tests were done with that configuration.

![Figure 5.10: Configuration summary](image)

Also if the median is a good distribution descriptor, it does not represent the whole distribution. And yet it would be important to get a better idea of how our data are distributed. For this we could use more statistic descriptors such as first, second (median) and third quartile. For this purpose we used box plot which is a helpful way of graphically characterizing groups of numerical data through their quartiles. In this representation, the thick line is the median, and a box is extending from the first to third quartile, and whiskers are extending to the maximum value below 1.5 time the interquartile distance. Other point are outliers plotted as individual points$^{14}$.

---

$^{13}$Internally the workload generator splits the number of TPS to generate to several processes using multiprocess library, for instance if the asked throughput is 900, there will be 30 processes generating 30 TPS each, this is due to the fact a single process can not generate a throughput on 900 TPS as the time it takes to get a response back from the database would be too long. That’s why we used this parallelization trick to make it possible.

$^{14}$We cut the graph after the whiskers.
5.6 Comparing shard key response time

5.6.1 Hypothesis

In the 6 shards configuration we compare performances of the two different shard keys with the new test protocol. Since sharding on property code enables query isolation, which means that query router can target directly the good shard(s) whereas sharding on hash-id implies contacting all shards (scatter-gather), we expect better performance when sharding on property code than hash-id.

5.6.2 Results

Figure 5.11 is the distribution of response time according to throughput.

We compute average response time (RT) in milliseconds when there was no workload and a workload of 1000 TPS, for the 2 different shard keys. For a given shard key we run a Student test\textsuperscript{15} between the average response time when there was no workload and a 1000 TPS one. Results are given in table 5.3.

\textsuperscript{15}t.test function in R.
Table 5.3: Response time comparison table between hash-id and property code shard keys strategy under an increasing workload

<table>
<thead>
<tr>
<th>Shard key/ Measure</th>
<th>RT no workload</th>
<th>RT at 1000 TPS</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property code</td>
<td>8.7</td>
<td>9.5</td>
<td>0.05196</td>
</tr>
<tr>
<td>Hash ID</td>
<td>9.3</td>
<td>30</td>
<td>0.00989</td>
</tr>
</tbody>
</table>

5.6.3 Interpretation

Firstly if we look at the median (thick bar), we find that for low throughput the median response time with the hash-id as shard key is lower than the response time when the shard key was the property code. The contrary happened for high throughput value. This is consistent with what we found with the first test protocol.

However if we look at the entire distribution (which is actually the most important) we found that for all throughputs values, sharding on the property code is far more better and scales better than the hash-id.

The Student test results also show that there was no difference (high p-value) of average response time with a generated workload of 1000 TPS or 0 TPS using the property code as a shard key unlike when the hash-id is used (low p-value). This confirms that property code scales better and our hypothesis. This is explained by the query isolation process we mentioned previously.

As a consequence, since our query contains the property code, if sharding is necessary choice of sharding on the property will be a good option to consider.

5.7 Shard on property code and high throughput

5.7.1 Hypothesis

In the 6 shards configuration we compare performance of the cluster when sharded on property code with an increasing throughput. Our hypothesis is that it should scale well.

5.7.2 Results

Figure 5.12 is the distribution of response time according to increasing throughput until 3000 TPS.
5.7.3 Interpretation

When the throughput increases the response time increases a little bit but the response time stays reasonable and under 20 ms. This increasing response time can be due to a beginning of database bottleneck\textsuperscript{16}.

5.8 Behavior under read/write workload

5.8.1 Hypothesis

In that test, we shard on the property code. The measuring process will only send search request to the database, therefore we continue to measure search response time. However the workload will not be constituted only of search operations, but of 25\% of write and the remaining of search operations, with the search queries being distributed as usual\textsuperscript{17}. Our hypothesis is that write operations are more expensive than read operations in particular due to IO access. Thus we expect worse but reasonable performances.
5.8.2 Results

Figure 5.13 shows response time distribution according to increasing throughput with two different workloads when the cluster was sharded on property code.

5.8.3 Interpretation

As imagined write traffic has an impact on response time but it is still under 20 ms.

5.9 Use two query router instead of one

5.9.1 Hypothesis

We are looking at ways to reduce response time. Our hypothesis is that by sending half of the queries to a second query router, that is to say a second MongoS should improve response time for high throughput.

---

16 Another reason (most likely not) that could explain an increasing response time could be explained by the fact the MongoS process starts to be overloaded or the machine running the MongoS itself. Experience with two MongoS will show it is not the MongoS but it could be the machine which is running the MongoS itself. It could also be explained by concurrency between MongoS and the measuring client.

17 Defined in 5.1.1.2.
Chapter 5. Performance tests

5.9.2 Results

Figure 5.14 is the response time distribution according to increasing throughput when sending queries to one and two MongoDB when the cluster was sharded on property code.

5.9.3 Interpretation

Sending queries to a second MongoDB on the same machine did not impact response time results, this might be due to the fact that we could not do much better\(^\text{18}\).

5.10 Conclusion

Given the workload, configuration and results we got in the single shard configuration we do not recommend to shard. Since this will add more complexity (increase response time) to handle a throughput higher than the one needed. The fact MongoDB is a NoSQL database does not mean we should absolutely shard our data. However the advantage is that later if the throughput needs to be increased it will be easy to scale with MongoDB. Here are some specific cases where it will be interesting to shard:

\(^\text{18}\)It would have been interesting to make similar tests with another MongoDB on another machine.
• If needed read/write throughput is increased significantly.

• If we need to distribute data across different locations as shown in 3.3.3 which might be a good solution for low read/write response time and increase data locality.

• Another good reason to shard is if the dataset (or working dataset) can not fit in memory. [38]. In our test any machine hosting the database had enough RAM to keep all the dataset in RAM alone since 30 million of booking is taking less than 100 GB of space. For an order of idea the average object size is 2032 bytes. MongoDB recommends to have at least the working set (last bookings) in memory.

We also strongly recommend to use replica set for high availability. If MongoDB is the master database, in order to guarantee consistency journaling should be enabled, and insertion should be done with journaling equal to true and with at least the "majority" write concern level.
Chapter 6

Search optimizations for Hotel Business

The goal of this section is to introduce some MongoDB optimizations specific to our use-cases. Firstly we will focus on MongoDB functionalities for matching, sorting and indexing in particular for search by last-name functionality. Then "home made" range of date and search by name optimizations will be introduced. The machine used for this test was equipped with 8GB of RAM and an Intel Core i5 CPU.\(^1\)

6.1 Functional study: Matching, sort and indexes capabilities study

6.1.1 Explain method

In order to evaluate query and index performances it is recommended to run the explain command on a cursor. This gives the query plan. In that query plan we focus on:

- n: the number of documents returned.
- nscanned: total number of index entries scanned.
- nscannedObjects: total number of documents scanned (when the information is not found in the index, the MongoDB engine follows a pointer from the index to the document to read the field directly)

\(^1\)This mainly explains why response time are higher. Actually during our test in this machine we observe many page faults, and the disk where data was located was not a SSD drive.
We will always have nscanned >= nscannedObjects >= n. The rule of the game is to try to reach a situation where nscanned and n are as close as possible (nscanned = n being the ideal case)

### 6.1.2 Basic sorting

#### 6.1.2.1 Index can help sort operations

MongoDB enables to sort by name and by date. For instance if we insert 200 documents containing a start date and a name, it is possible to sort all documents by names as shown in the example below:

```python
1 # doc pattern in database : {"date": start_date, "name": name}
2 queryDict = {}
3 projDict = {"_id":0, "name": 1}
4 cursor = db.collection.find(queryDict, projDict).sort([("name", pymongo.ASCENDING)])
```

In this query no index is used, all the database is scanned and sorted into memory. If we run an explain on that query the scanAndOrder field will therefore be set to True. If more documents are inserted into the database, MongoDB won’t be able to sort into memory all the documents in the database, sort operations can not consume more than 32 MB, and the previous query will lead to the following error message.

```python
1 pymongo.errors.OperationFailure: database error: Runner error:
2 Overflow sort stage buffered data usage of 33554472 bytes exceeds internal limit of 33554432 bytes
```

So as to be able to sort by name with a bigger dataset or make the previous query faster, we can limit the number of results to be returned (note that sort is done before limit), or index the field we want to sort on. The optimal case being a combination of the two techniques. Obviously when indexes are created and/or the number of results is limited to the first hundred the response returned by the database is exactly the same as the first hundred documents returned in the previous example. When we run an explain on that query, scanAndOrder is False, since MongoDB did not have to sort the results, since it used the already sorted index. Also since we only project the name field when indexes are created nscannedObject is equal to 0, as the query can be answered by reading the index only.
6.1.2.2 UTF8 sort

MongoDB does not enable to sort by local and sort order will be based on UTF8 order. For instance Swedish letters ö, ä, å will be ordered as ä, å, ö (UTF8\(^2\)) instead of å, ä, ö (Swedish alphabet order).

6.1.3 Array, matching, sorting and impact on document modeling

Suppose we insert the following documents into the database with a structure similar to the one there is in our document modeling. Each booking contains a list of guests names on which we want to be able to make search and sort operations. It is necessary to learn more about how this structure can be queried.

```
1 { "guest_names": [
2     { "last_name": "besson",  "first_name": "luc" },
3     { "last_name": "lucas",  "first_name": "georges" }
4 ]},
5 { "guest_names":
6     [  
7         { "last_name": "hitchcock",  "first_name": "alfred" },
8         { "last_name": "kubrick",  "first_name": "stanley" }  
9     ],
10 ]},
11 { "guest_names":
12     [  
13         { "last_name": "chaplin",  "first_name": "charlie" },
14         { "last_name": "eastwood",  "first_name": "clint" }  
15     ],
16 ]},
17 { "guest_names":
18     [  
19         { "last_name": "truffaut",  "first_name": "francois" }  
20     ]
21 }
```

6.1.3.1 Matching

Single field matching It is possible to find an element in an array, for instance the following query:

```
1 {"guestnames.last_name": "chaplin"}
```

will return the third document. Actually the array is encoded as a document [51] as shown below:

\(^2\)In UTF8: "ä, å, ö" = "xe4, xe5, xf6".
Thus it is also possible to query on an element in an array and the following query is legal:

```
{ "guest_names.1.last_name" : "chaplin" }
```

But it will not return any document since the value "charlie" is associated to the key 0. However

```
{ "guest_names.0.last_name" : "chaplin" }
```

will return the third document.

If an index is created on "guest_names.last_name", the query

```
{ "guest_names.last_name" : "chaplin" }
```

will be able to use the index efficiently (nscanned = 1) however "guest_names.0.last_name" : "chaplin" and "guest_names.1.last_name" : "chaplin" won’t use the index efficiently (nScanned = 7). However it is also possible to create an index on "guest_names.0.last_name" or "guest_names.1.last_name".

**Sort**  Similarly to the matching part it is possible to sort on the array, if a sort is applied on "guest_name.last_name" and we project only the last name, the document will be ordered as it is shown in the following output:

```
{u'guest_names': [(u'last_name': u'besson'), (u'last_name': u'Lucas')]}
{u'guest_names': [(u'last_name': u'chaplin'), (u'last_name': u'Eastwood')]}
{u'guest_names': [(u'last_name': u'hitchcock'), (u'last_name': u'kubrick')]}
{u'guest_names': [(u'last_name': u'truffaut')]}  
```

The sort is done on min value of last_name for each document. Similarly to the matching mechanism it is also possible to sort on a specific value in the array, for instance a sort by "guest_names.1.last_name" will order the documents with the right value of the array as follows:
As previously if an index is created on the "guest_name.last_name" field, a sort by "guest_name.0.last_name" and "guest_name.1.last_name" will not use it efficiently (scanAndOrder is true) unlike a sort by "guest_name.last_name". However it is also possible to index sub element of an array³.

### 6.1.3.2 Comments

If the system should be able to match any guest name and sort by the lowest name, this data model fits well, and an index on the whole array should be created. That is what has been done for the POC and tests.

However a hotel booking contains customers who can be the booker and simple occupants or both at the same time (see in appendix A is_booker and is_occupant fields), hence if it is only needed to match the booker name and not all the occupants name, and sort by the booker’s name, an index on a element of the subarray should be created. In that case it might be cleaner to put booker in a sub-document and all occupant in a array as we did. That way we won’t have to index the whole guest name array and thus this will avoid index unnecessary data. As we can see here this is an example where queries impact modeling.

These two scenarios are easy ones because the matching part is on the same field as the sort. But if the system should be able to match any customers names and sort by booker name, then an index on the whole array or a specific element or a combination of both should be created. The advantage being given to the matching or sorting step. In that case the query is as if the matching part was on a different field as the sort. Index strategy for that kind of request will be studied later (in 6.1.4.2) and can apply well for that case.

### 6.1.3.3 Matching on two documents fields

In this section we will study what happens if we match two fields in the same documents. Here are the query JSON documents (db.bookingCollection.find(jsonDoc)) which were executed:

³Such as "guest_name.0.last_name" and "guest_name.1.last_name".
We observe the following results:

- Case 1: will return the third document as this is a document for which last name and first name match.

- Case 2a: will NOT return any document because there are no documents containing these two names at the same time.

- Case 2b: will return the third document, since this document has "chaplin" as a last name and "clint" as a first name, even if they are not belonging to the same person.

- Case 3a: will NOT return any documents because we now force that last name and first names belong to the same person.

- Case 3b: will return the third document as this document has a guest whose last name AND first name are equal to the one in the query.

### 6.1.4 Compound index

A common query is a search by check-in between two dates and a check-out between two dates for a specific hotel as it was introduced in the prototype section. This search contains several fields, for good performance on such queries MongoDB can use a Compound index. While the order of keys in the query does not have importance, the order of keys in the index matters and it is worth to study them more deeply.

Some general theory, compound index can help for search, if the collection contains an index on \{a:1, b:1, c:1, ... n:1\}, we say that \{a:1\}, \{a:1, b:1\} ... \{a:1, b:1, ... n:1\} are prefixes of this index. This index will enable to perform efficiently queries containing the index or prefixes, for instance: \{a:1\}, \{a:1, b:1\}, \{a:1, b:1, c:1\}. However a query, which does not contain the index or its prefixes such as \{b:1, c:1\}, will not be able to use this index efficiently. The sort operation can also take advantage of compound index, if the index is organized in the same order as we want to sort.
According to this element of theory some experiments have been realized so as to determine the best indexing strategy for different use-cases that could be met in the reservation search use-cases. We will take as an example the search by check-in and check-out date. This search is a range query and not an exact match. The results found here can also be applied to a search by name, since they might be also range query as we will see later.

### 6.1.4.1 Key order in compound index

An important question is the order of key in a compound index. The main idea is that the first key should reduce the search space as much as possible. A question we have been wondering when creating a compound index on property code, check-in and check-out date is if we should create an index with the property code as a first or last key as shown below:

1. Index 1: `{ "proptyIDs.propty_code" : 1, "room_stay.dates.begin" : 1, "room_stay.dates.end" : 1 }`
2. OR
3. Index 2: `{ "room_stay.dates.begin" : 1, "room_stay.dates.end" : 1, "proptyIDs.propty_code" : 1 }`

For determining the best method we made benchmarks with the two indexing strategies, when the query was on an exact hotel (property code) and a range of check-in date and check-out date (we always search for a range of date within a property). Results are shown in table 6.1.

<table>
<thead>
<tr>
<th></th>
<th>Index 1</th>
<th>Index 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>nScanned</td>
<td>831</td>
<td>1950</td>
</tr>
<tr>
<td>nScannedObject</td>
<td>831</td>
<td>831</td>
</tr>
<tr>
<td>n</td>
<td>831</td>
<td>831</td>
</tr>
<tr>
<td>millis</td>
<td>9</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 6.1: Index order strategy, Property is the first key in index 1 and last key in index 2

The index starting by check-in date (index 2) is slower because the number of documents scanned is higher (nscanned). This is because when the index starts by check-in date and check-out date, it matches a set of document which fall in the correct range of date. Then inside this first set of documents it is needed to discard documents where the property code is not the one mentioned in the query. Therefore MongoDB needs to read more index entries than just those returned and is slower. However if we use the property code, the query engine will be able to target directly the right property and within that property, access directly to the documents which fall in the correct range of date. That is why we are in the ideal case where nscanned = nscannedObjects = n. A conclusion from this test is that for
range query, we should put first in the compound index the field which will be used for an exact match. It is important to notice here that if the query was not an exact match on property code, check-in and check-out date, it might have been better to put first the index with highest cardinality first, to reduce the search space. In our test, for an exact match on property code, no difference was found between the two orders.

Another main reason it is better to index by property code first, is also for sharding, as we saw that sharding on that compound index had interesting properties.

### 6.1.4.2 Sort and compound indexes

**Sort is done on the same key as query field**  In the previous section we determined that an index starting on property code was optimal for range date. It is also possible to sort the result based on a specific field, the sort requirement might impact the index strategy. Suppose we want to sort by check-in date. This can be done by applying the sort operator on the cursor like `cursor.sort([("room_stay.dates.begin", pymongo.ASCENDING)])`. The sort method takes as the parameters an array containing the fields we want to sort by and the sort order (ascending or descending)\(^4\). It is also possible to sort on several fields, for instance it is possible to sort by check-in date and check-out date simultaneously\(^5\).

Using the compound index with property-code as a first key, followed by check-in date and then check-out, we could run a query of range of date for a specific hotel and sort results by "check-in date" or "property code and check-in date", and by "check-in date and check-out date"\(^6\). For that query MongoDB will be able to use indexes to match (nscanned=nscannedObject=n) as previously and sort (scanAndOrder = False) the results efficiently. Indeed the fields we "sort by" are the same as those in the query, hence sorting is done at no cost since MongoDB just needs to read the index which is already well ordered. It has also been noticed there was no difference between sorting by property code AND check-in date or by only sorting by check-in date, because our query matches a range of dates within a given property. Nevertheless if the query had targeted a range of property codes, the sorting by check-in date only would have needed an in memory sort unlike the sort on property code AND check-in date, to re-order the result by check-in date, as they would

\(^4\)Sort parameters are a JSON document in native API and an array like here in Python API, ASCENDING is equal 1 and DESCENDING to -1.

\(^5\)By doing `cursor.sort([("room_stay.dates.begin", pymongo.ASCENDING), ("room_stay.dates.end", pymongo.ASCENDING)])`.

\(^6\)Sorting by "check-out date and check-in date" or "check-out date" only would not have used the index efficiently.
have been naturally ordered by property code AND check-in date. Sort on the same fields as the one we are querying on is the most used scenario and confirms the index strategy proposed.

**Sort is done on field that are not in the query** It is possible to sort on field(s) which are different from the one used in the matching part of the query. However this might change the index strategy. In appendix C, we gave more details on that point.

**Sort on two different fields** It is also possible to sort on several fields. We give also more details and an indexing strategy for that case in appendix C.

### 6.2 Search optimizations

Here we introduce some search optimizations for hotel business.

#### 6.2.1 Use of geospatial indexes to find documents which fall into two ranges of dates

MongoDB has a special feature called Geospatial index. A booking is a document which contains two dates: a begin date and an end date. We would like to be able to find all reservations which begin date is between two dates and end date is also between two dates. As done in the POC, previously we have used MongoDB compound indexes, thus we indexed start date and end date field, and use "gte" and "lte" operators to find all documents which were into the two ranges of date. However we have been wondering if we could improve query performances by using GEO2D indexes. For this we could convert begin date and end date to unix time, then each booking is a point whose position is (start date, end date). Using the $within operator it makes it possible to get documents that are within the bounds of the rectangle specified by bottom left and upper right coordinate. Therefore finding bookings which begin date is between a check-in start and check-in end date and end date is between a check-out start and check-out end date is equivalent to find all points in the box specified by bottom coordinate (check-in start, check-out start) and upper coordinate (check-in end, check-out end) as shown in figure 6.1.

We will compare performances of these two indexing strategies. In appendix E, an implementation comparison between these two protocols is given.
During our test we realized there was no performance improvement when using geospatial indexes. Actually compound indexes perform better in terms of response time. This is due to the nature of our query and data. In our test the distribution of date in query and data were the ones described in 5.1.1.2. In that particular case the range of check-in and check-out in search queries are small (<31 days). The room-stay are also small. Therefore a compound index performs really well because bookings which are selected by the first key of the index (check-in date) are also those which match the second part of the index (check-out date). Therefore even if nscanned > n, nscanned is close to n. However, if we make a query with very large check-in (several years) and a small check-out (one day), we identified that a geospatial index might do better. This is due to the fact that the majority of documents selected by the first part of the compound index will be rejected by the second part of the index, thus nscanned >> n. In that case a compound index is worse than a 2D index. Regarding our data and query, it is better to keep the compound index strategy.

In terms of implementation, geo indexes in MongoDB are using geohash. Geohash is a mechanism which converts a two dimensional value to a single dimension one by keeping proximity information. Then this Geohash is indexed in a B+Tree index. The

---

7If we run an explain on a range of check-in and check-out date query, when using the compound index on check-in and check-out fields we will always have nscanned > n whereas when using the 2D index we will have nscanned = n. However the response time is slower when using the 2D index than the compound index on check-in and check-out out fields.
results might have been different if geospatial index would have been implemented using RTree indexes instead of B+Tree indexes\(^8\).

Note It would have also been possible to create a compound index, containing a 2d index/field. For an instance a Geospatial index as a first key and property code as a second key. The contrary is actually not possible, 2d key should always be the first unlike traditional indexes. It is also not possible to shard on a geospatial index key. Previously queries were only on range of check-in and check-out date, but this could have helped if we had included the property code in the query.

However we choose not to compare a compound index containing keys on property code, check-in and check-out date with a compound index containing a 2D key and property code, because we wanted to evaluate search by date performance alone using the two methodologies. However we should keep in mind that using the property code would have advantaged the non GEO index method, as the property could have been used as a first key, and it would have helped the query by reducing the search space as shown previously. Also as it is not possible to shard on a 2D index and query isolation using the property code would not have been possible since the 2D field should be the first key. Lastly it will be easier for the original strategy to return sorted results by "check-in" or "check-in and check-out date". All these facts lead also to prefer the original strategy.

### 6.2.2 Search by name(s) optimizations

#### 6.2.2.1 Problem description

Previously we have seen it was possible to search by name by doing an exact match or by a mentioning the first letters of the name (start with) using the $regex operator. By analyzing the last name field, we observe that some last names were composed of several names (for instance Spanish names) and that often travel agents fill the last name field with the first name followed by the last name. So suppose the last name field contains "don diego de la vega". If we want to find a booking associated to this name, and search booking whose guest name is "vega" or start with "veg", the reservation system will never be able to find this booking. Nevertheless this should be possible. In this section we will introduce several ways to solve this problem which has been found using MongoDB and the document approach and will benchmark the different solutions\(^9\).

\(^8\)Note that beta version of Couchbase geospatial index (GeoCouch) is actually using RTree indexes.

\(^9\)Also to simplify the problem, we store the last name field as a simple field and not an array of names.
6.2.2.2 The three methods

Using $regex We store name in a field called last name, create a single field index on last name and use the $regex operator to find if the string in the query is contained in the last name field of one of the document in the collection.

Using Text index MongoDB supports full text search. Thus as previously we use the same last name field except we will index it as text indexes. Then we use the "text" operator to make a search on this index to find if one of the name in the query is in the document.

Using array indexes The idea here is to split the different components of the last name field into an array, for instance the name "diego de la vega" will also be stored as the following array in the document ["diego", "de", "la", "vega"]. Then we index this field as a multikey index\(^\text{10}\). Therefore when making a search in the array, if one of the terms in the query is the same as one in the name and thus in the array, the document will be returned.

Synthesis In array 6.2, we gather the three proposed strategies. We made requests to the database by applying the find function to the booking collection using the query document as parameter. We also create an index by using the ensure_index method on the collection with the index creation array\(^\text{11}\).

<table>
<thead>
<tr>
<th>Document to match field</th>
<th>Regex</th>
<th>Text indexes</th>
<th>Array indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ &quot;last_name&quot;: &quot;don diego de la vega&quot; }</td>
<td>{ &quot;last_name&quot;: &quot;don diego de la vega&quot; }</td>
<td>{ &quot;names&quot;: [&quot;don&quot;, &quot;diego&quot;, &quot;de&quot;, &quot;la&quot;, &quot;vega&quot;] }</td>
<td></td>
</tr>
<tr>
<td>([&quot;last_name&quot;, 1])</td>
<td>([&quot;last_name&quot;, &quot;text&quot;]):</td>
<td>([&quot;names&quot;, 1])</td>
<td></td>
</tr>
<tr>
<td>{ &quot;last_name&quot;: { &quot;$regex&quot;: &quot;vega&quot; } }</td>
<td>{ &quot;$text&quot;: }</td>
<td>{ &quot;names&quot;: &quot;vega&quot; }</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: The three indexes strategy

\(^\text{10}\)In MongoDB, create index on a single field index or multikey index follows the same syntax (same for search), this can be confusing.

\(^\text{11}\)In Pymongo the ensure index method takes as parameter an array as the one used for the sort operation, it is also a JSON document in the native API.
6.2.2.3 Benchmarking the three methods

We measured median response time for these three configurations. As shown in figure 6.2, $regex did not perform well at all. We made a zoom in figure 6.3 in which we can determine that in terms of response time for this specific purpose array indexes perform better than text indexes.

6.2.2.4 Sort results by relevance

We will now focus on the text and array indexes strategy. Suppose that it is now possible to specify several names as a search parameter in what we call a search string (like the one in a search engine). In that case we might want to sort results by relevance. For instance if search parameters are "diego vega zorro", it should return the most relevant documents first. For instance if there is a document whose last name field is "diego de la vega" and a second one where it is "sylvain de la vega" in
Chapter 6. Search optimizations for Hotel Business

Figure 6.4: Comparison of text indexes and array indexes (aggregation) response time when sorting results by relevance

For a collection, for that query the first one is more relevant. When using text indexes, MongoDB computes a matching score and thus it enables to sort results by relevance. For array indexes there is no direct solution. However this remains possible by using the aggregation framework in a particular way. This way was explained by John Page on his blog[52] and also enables to sort by score. For instance if "diego vega zorro" is the search string, the score of a document containing "diego de la vega" will be 2, and the one containing "sylvain de la vega" will be 1. Hence sorting by this score field enables to sort by relevance. In appendix D an implementation of these two methods is given.

6.2.2.5 Benchmark with a sort constraints

As shown in figure 6.4, in that case the text indexes performs more efficiently. Also the response time is lower than in the previous test because we only project the id field.

6.2.2.6 Going further

Store all names related to a booking in a text or array index Following the same methodology a "last names" field could contain all guest names. So that the document with the most common names with the query string will be returned first.

Phonetic search There is an algorithm called "double metaphore" (similar to soundex) which is a phonetic algorithm and enables to index names by sound. For
instance "coulombel" and "coulombelle" will be indexed as "KLMPL" string. Therefore it make it possible to find the booking whose customer’s name is "coulombel" even with an incorrect writing. For this purpose we also store in the names string of the document, the double metaphone string of each name. Similarly in the query string the double metaphone string of each name has also been added. That way the result will contain documents with the correct written name but also those which do not have the correct spelling but a similar sounding. The advantage being that documents with same spelling (and thus same sounding) as the one in the search string will have a better relevance and therefore will be displayed before the one which have only a similar sounding. Also note that a major issue with phonetic algorithm is that they are language dependent although the double metaphone algorithm is working with many languages.

**Weight on text indexes**  Note that the text index created previously was on one field and not the whole document. If only one text index is allowed per collection (same for geospatial index) it is possible to index several fields in the same text index. This makes it possible to even index all documents fields in the text index. An interesting point is that a weight can be specified for each field. This can make an interesting use, for instance we could index booker name with more weight than other customers names.

### 6.3 Conclusion

This section gave an insight of more complex search functionalities and enables to discover that MongoDB was able to perform even more complex queries.
Chapter 7

Chapter conclusion

7.1 Conclusion

As a conclusion it appeared during the project that a document store is an excellent choice for storing hotel reservations among the NoSQL solutions (Chapter 2).

Between the two main document stores MongoDB has been chosen over Couchbase for its better handling of documents. However we have seen that MongoDB does not support master-master replication in particular across two data-centers unlike Couchbase (XDCR). However shard and replicas can be distributed in two data-centers. MongoDB can guarantee different levels of consistency\(^1\) at the document level in write and performs consistent write\(^2\) and read\(^3\). MongoDB replica set provides high availability and automatic failover, however in case an entire data-center is down, a part of the data might be available only on read access\(^4\) (Chapter 3). This lack of cross data-center replication can be a showstopper if two data-centers are needed in different continents to provide low latency (reduce network cost) and also ultra available read and write even in case of an entire data-center failure.

Scaling vertically is possible using sharding, and might be useful to increase maximum throughput or available RAM. However scaling vertically has a cost, it adds more complexity and increases the response time. With a single shard, our dataset entirely fits in RAM and we were able to already reach a very high throughput. It means that although MongoDB is a NoSQL database, sharding decision should not

---

\(^1\)Multi data-center consistency has been discussed in chapter 3 in paragraph 3.3.3.3.
\(^2\)If w = majority, j = true, journaling enabled, note that as master-master replication is not permitted thus there is no write conflict.
\(^3\)consistent read if reading is not activated on replicas, in Couchbase views are usually asynchronous and returns stale data.
\(^4\)Couchbase actually handles better this case.
be automatic. Except the case of sharding for distributing data geographically as introduced in chapter 3, it might be good not to shard in a first time, knowing that it will be possible to do it so as to increase the maximum throughput or RAM. When sharding it has been shown that for hotel use-cases, sharding on the property code would be a really efficient strategy, since it would enable the MongoDB engine to make targeted requests instead of scatter-gather one. The response times to answer the queries are promising (Chapter 5). Therefore search performances are very satisfying.

Finally we have seen that the MongoDB data manipulation language (find, aggregate and map-reduce) enables to fill basic search functionalities as it was demonstrated in the POC (Chapter 4) and make even more complex queries taking advantage of the document structure (Chapter 6). Hence complex search functionalities can be implemented too with MongoDB.

7.2 Future work

During this work we try to gather key points to consider when moving from the relational world to NoSQL technologies. If our prototype and benchmarks proved us, that using document store and in particular MongoDB for storing and search of reservations was possible, there will be always something more to do. Indeed MongoDB offers many possible options which can be combined. This often complicates our tasks, because we would have liked to try more configurations and it makes it hard to focus on a set of specific features for the study. It could be interesting to evaluate more deeply replica set and also how we could geographically distribute data in practice.

Also it is important to stay up-to-date as this domain is evolving fast\textsuperscript{5}, and what is true today might be false tomorrow. Maybe in the future MongoDB will enable master-master replication? Couchbase is working on N1QL, a SQL like language for querying Couchbase, thus we can wonder if one day Couchbase will support better document handling than MongoDB?

\footnote{\textsuperscript{5}For instance during our work, there was a new release of MongoDB (2.6) with new features such as official support of text index and index intersection.}
Appendix A

A booking example

Here is given an example of a booking stored in the database:

```json
{
  "IDs": {
    "CAN_NUM": "VTNV3 W T 2CD",
    "CF_NUM": "XJZ7 W Q F D 8T",
    "PNR": "JQ2ZGJ9U6C"
  },
  "T_data": {
    "T_date": {
      "$date": 1358899200000
    },
    "T_status": "CANCEL"
  },
  "comments": [
    "non smoking",
    "sea view"
  ],
  "customers": [
    {
      "first_name": "francois",
      "id": 0,
      "is_booker": "True",
      "is_occupant": "True",
      "last_name": "gomez",
      "mail_adress": "francois.gomez@yahoo.com",
      "original_first_name": "francois",
      "original_last_name": "gomez"
    },
    {
      "first_name": "pierre",
      "id": 1,
      "is_booker": "False",
      "is_occupant": "True",
      "last_name": "renard",
      "original_first_name": "pierre",
      "original_last_name": "renard"
    }
  ]
}
```
"form_of_payment": {
    "credit_card": {
        "card_expiry_date": {
            "$date": 1422489600000
        },
        "card_vendor_code": "MC",
        "holder_name": "gomez"
    }
},
"global_amount": {
    "amount_after_taxes": 840.0,
    "amount_before_taxes": 700,
    "currency": "EUR"
},
"originator": {
    "amadeus_office_id": "BFLZ1291",
    "amadeus_office_id_auth": "WMDB101"
},
"proptyIDs": {
    "brand_code": "BL",
    "chain_code": "DHM",
    "propty_city": "REO",
    "propty_code": "BLPARP04"
},
"random": 0.8739503682800664,
"room_stay": {
    "customers": [
        0,
        1
    ],
    "dates": {
        "begin": {
            "$date": 1386892800000
        },
        "end": {
            "$date": 1387238400000
        }
    },
    "rate_plans": {
        "rate_plan": {
            "begin_date": {
                "$date": 1386892800000
            },
            "end_date": {
                "$date": 1387238400000
            }
        },
        "services": [
            {
                "code": "BRK",
                "name": "PETIT DEJEUNER"
            }
        ]
    }
}
Appendix A. A booking example

```json
}
{
  "services": [
    {
      "code": "JAC",
      "name": "JACUZZI",
      "service_dates": {
        "begin_date": {
          "$date": 1386892800000
        },
        "end_date": {
          "$date": 1387238400000
        }
      }
    }
  ]
}
```
Appendix B

Different kinds of indexes existing in MongoDBs

Here are the 5 kinds of indexes supported by MongoDB:

- **Single Field Indexes**: "A single field index includes data from a single field of the documents in a collection. The field being a field at the top level of a document or in sub-documents"\(^1\) [33, indexes]. The index can be for instance \{a:1\}, this will index the field `a` in ascending order.

- **Compound Indexes**: "A compound index includes more than one field of the documents in a collection" [33, indexes]. While the order of keys in the query does not have importance, the order of keys in a compound index matters. For instance if the collection contains a compound index on \{a:1, b:1, c:1, \ldots n:1\}, we say that \{a:1\}, \{a:1, b:1\} \ldots \{a:1, b:1, \ldots n:1\} are prefixes of this index. This index will enable to perform efficiently query containing the index or prefixes, for instance: \{a:1\}, \{a:1, b:1\}, \{a:1, b:1, c:1\}. However a query, which does not contain a field such as \{b:1, c:1\}, will not be able to use this index. Compound index can also enable to sort efficiently, if the index is organized in the same order as we want to sort.

- **Multikey Indexes**: "A multikey index (array index) is an index on an array field, adding an index key for each value in the array"[33, indexes]. If "a" is in array, \{a:1\} will index the array. Note there is no difference with single field indexes creation syntax.

\(^1\)The field can be an entire sub document
Appendix B. Different kind of indexes existing in MongoDB

- Geospatial indexes (B+tree based on Geohash). For instance if the document contains a field which is position : [long:x, lat:y], {pos:"GEO2D"} will create a geospatial index.

- Text Indexes : "search of string in indexed field" [33, indexes]. If a is a field containing a string, {"a":"text"} will create a text index on a field.

- Hashed Index : "Hashed indexes maintain entries with hashes of the values of the indexed field" [33, indexes]. For example {"_id": "hashed"}, will create an index on hash of the id.
Appendix C

Complex sort and compound indexes

C.1 Sort is done on fields that are not in the query

Let’s suppose a requirement is to sort check-in date results by card holder name. In that case the sort clause will be:

```python
.sort([("form_of_payment.credit_card_holder_name", pymongo.ASCENDING)])
```

Two different indexing strategies have been tried: index 1 will give the advantage to document matching (using same index as in 6.1.4.2) and index 2 to sorting (using an index on card holder name):

1. Index 1:
   ```
   { "proptyIDs.propty_code" : 1, "room_stay.dates.begin" : 1, "room_stay.dates.end" : 1 }
   ```

2. OR

3. Index 2:
   ```
   { "proptyIDs.propty_code":1, "payment.card.holder_name":1}
   ```

Results are shown in table below:

As we can see the first index was faster, and it only scanned the returned object unlike the second index which had to scan 6242 objects (all documents within the specified property). However index 2 did not have to sort results in memory (scanAndOrder is False) unlike index 1 as it reads the index directly in the right order. The

<table>
<thead>
<tr>
<th>Explain key</th>
<th>Index 1</th>
<th>Index 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanAndOrder</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>nScanned</td>
<td>831</td>
<td>6242</td>
</tr>
<tr>
<td>nScannedObjects</td>
<td>831</td>
<td>6242</td>
</tr>
<tr>
<td>n</td>
<td>831</td>
<td>831</td>
</tr>
<tr>
<td>millis</td>
<td>33</td>
<td>149</td>
</tr>
</tbody>
</table>
in memory sort consume database resources and should be avoided, but we have shown here it was faster to sort results in memory than scanning more documents. As a consequence index 1 is still the best.

Then we repeat the same experiment, but this time the number of results to return will be limited to the first hundred. In that case index 2 performs better. Therefore advantaging sorting might be a good idea when limiting the number of results\(^1\)

### C.2 Sort on two different fields

It is also possible to sort on two different fields, for instance it is possible to sort by name and check-in date and out date by doing:

```python
cursor.sort([("payment.card.holder_name", pymongo.ASCENDING), ("room_stay.dates.begin", pymongo.ASCENDING)])
```

Each key can be ordered ascending and descending, thus we can also for instance sort by ascending name and descending begin date. However if we create an index ascending on name and date this index won’t be able to use it to sort efficiently\(^2\). Indeed unlike a query with a single field sort using a single field index where key order does not matter as MongoDB can read the index in two order, it is an important point to be taken into consideration here. If the functionality to index in the two orders is needed, it will be necessary to create two different indexes to perform it efficiently. However note that an index for instance on ascending date and descending name can be used to sort on descending date and ascending name (multiply by -1 order value).

\(^1\)In MongoDB 2.6 when running the query with index 1 and limiting the results, we got strange explain output the cursor used was “QueryOptimizer” and “scanAndOrder” was false, whereas it was true using 2.4 version, and which is more logical. Some recent change might have been made. We got more info on queryOptimizer here [https://jira.mongodb.org/browse/SERVER-14585](https://jira.mongodb.org/browse/SERVER-14585) from MongoDB. But index 2 performs better when there was a limit AND if we limit the result to the first k ones (limit k), the number of documents return without the limit is greater than k.

\(^2\)scanAndOrder is true in 2.4 version but as the case below in MongoDB 2.6, query optimizer cursor is used and scanAndOrder is False, but this query takes longer.
Appendix D

Search by name(s) optimizations implementation

D.1 Text index and sort by score

```python
family_names = "diego vega"

db.bookingCollection.ensure_index(["last_name"], name = "search_index")
queryDict = { "$text": { "$search": family_names}}

cursor = db.bookingCollection.find(queryDict,
    { 'score':
        { '$meta': 'textScore' }, "_id":1}
    )
.cursor(1
    .sort([('score', { '$meta': 'textScore'})])
    .limit(limit_value)
```

D.2 Array index and sort by score using the aggregation framework

```python
family_names = "diego vega"
family_name_split = family_names.lower().split(" ")

for e in family_name_split:
    orArray.append("customers.all_names": e )
    innermatchesArray.append("$cond"::[ {"$eq" : ["$customers.all_names", e ]},e.e,""])
```
query = {
    "$or" : orArray
}

aquery = { "$match" : query}

aproyect = {"$project" : {"customers.all_names":1}}

aunwind = {"$unwind" : "$customers.all_names"}

aproyectX = {"$project" : {"c" : { "$concat" : innermatchesArray}, "_id" : "$id"}}

amatchX = {"$match" : {"c" : {"$ne" : ' '}}}  

agroup = {"$group" : {"_id" : "$_id", "c" : {"$addToSet" : "$c"}}} 

agroup2 = {"$group" : {"_id" : "$_id", "c" : {"$sum" : 1}}} 

asort = {"$sort" : {"c" :-1} }

alimit = {"$limit":limit_value}

res = db.bookingCollection.aggregate([ 
aquery, aproject, aunwind, aproyectX, amatchX, agroup, aunwind2, agroup2, afilter, asort, alimit 
])

result = res['result']
Appendix E

Search by 2 ranges of date implementation

In this appendix is given (compared) the implementation of a compound index on start and end date field with a GEO2D index for search by two ranges of date.

E.1 Room stay dates in the document

E.1.1 Compound index

Date field in a document inside the database.

```
"room_stay" : {
    "dates" : { // This is used by compound index
        "begin" : ISODate("2018-09-03T00:00:00Z"),
        "end" : ISODate("2018-09-28T00:00:00Z")
    }
}
```

E.1.2 Geo2D index

```
"room_stay" : {
    "geoDates" : [
        // This is a begin and end date timestamp used by 2D indexes
        1535925600,
        1538085600
    ]
}
```
Appendix E. Search by 2 ranges of date implementation

### E.2 Index creation

Here are how the two kinds of indexes are created.

#### E.2.1 Compound index

```python
db.bookingCollection.create_index(
    [
        ("room_stay.dates.begin", pymongo.ASCENDING),
        ("room_stay.dates.end", pymongo.ASCENDING)
    ]
)
```

#### E.2.2 Geo2D index

```python
db.bookingCollection.create_index(
    [
        ("room_stay.geoDates", "2d")
    ],
    min = 1200000000, // min boundary
    max = 80000000000 // max boundary
)
```

If documents contain coordinate data outside of the specified range, MongoDB returns an error.

### E.3 Data querying

Here are how the two kinds of indexes are queried (db.bookingCollection.find(queryDocument))

#### E.3.1 Compound index

```python
queryDocument =

{
    'room_stay.dates.end': {
        '$lte': datetime.datetime(2012, 2, 9, 0, 0),
        '$gte': datetime.datetime(2012, 1, 28, 0, 0)
    },
    'room_stay.dates.begin': {
        '$lte': datetime.datetime(2012, 2, 7, 0, 0),
```
Appendix E. Search by 2 ranges of date implementation

```javascript
queryDocument =

{ 'room_stay.geoDates':
  { '$within': {
    '$box':
      [ [1327532400.0, 1327705200.0],
       [1328569200.0, 1328742000.0]
  ]
}
}
```

E.3.2 Geo2D index
Bibliography


[41] Mongodb google groups : Data loss and missing acknowledgements in different scenarios with mongodb, 2014. URL https://groups.google.com/forum/#!msg/mongodb-user/SDY82Vzfif0/h4Z2ZIEY0ikJ.


Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Sylvain COULOMBEL.