Evaluation of Major NoSQL Databases

George Antoniou

August 19, 2016
Abstract

Since the invention of the relational model in 1970, relational databases have been the most popular means of storing and managing data. However, recently NoSQL databases have been gaining popularity. In this dissertation project three major NoSQL databases, namely MongoDB, Cassandra and Amazon DynamoDB are evaluated qualitatively the former two are evaluated quantitatively as well. Instances of MongoDB and Cassandra were installed on a single node and on a multi-node cluster, and their main features were investigated and their performance was assessed. DynamoDB is a Database as a Service (DBaaS) that is hosted within the Amazon Web Services (AWS), and therefore, its cloud-related features were investigated. The three databases were compared with each other using a set of quality attributes and performance metrics. The comparison results indicate that the three databases have different strengths and weaknesses; each of them excels at solving specific type of problems. At the end of this project suitable use cases for each database are recommended and general usage guidelines are proposed.
# Contents

Chapter 1 Introduction ........................................................................................................... 1  
1.1 Dissertation purpose ........................................................................................................ 1  
Chapter 2 Background Theory ............................................................................................... 4  
2.1 Big Data .......................................................................................................................... 4  
2.2 Relational databases ......................................................................................................... 5  
  2.2.1 Database normalisation .............................................................................................. 5  
  2.2.2 ACID transactions ...................................................................................................... 6  
  2.2.3 Limitations ................................................................................................................ 6  
2.3 NoSQL Databases ............................................................................................................. 7  
  2.3.1 Key-value stores ........................................................................................................ 7  
  2.3.2 Document stores ....................................................................................................... 8  
  2.3.3 Wide-column stores .................................................................................................. 9  
  2.3.4 Graph stores ............................................................................................................. 9  
2.4 Related work .................................................................................................................... 10  
  2.4.1 Quantitative evaluations ......................................................................................... 10  
  2.4.2 Qualitative evaluations ............................................................................................ 11  
Chapter 3 MongoDB qualitative evaluation .......................................................................... 12  
3.1 Data model ...................................................................................................................... 12  
3.2 Architecture .................................................................................................................... 12  
  3.2.1 Auto-sharding ........................................................................................................... 12  
  3.2.2 Availability, Consistency and Durability .................................................................. 13  
3.3 Data import ..................................................................................................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Query language</td>
<td>16</td>
</tr>
<tr>
<td>3.4.1 Create</td>
<td>16</td>
</tr>
<tr>
<td>3.4.2 Read</td>
<td>17</td>
</tr>
<tr>
<td>3.4.3 Update</td>
<td>17</td>
</tr>
<tr>
<td>3.4.4 Delete</td>
<td>18</td>
</tr>
<tr>
<td>3.4.5 Text search</td>
<td>18</td>
</tr>
<tr>
<td>3.4.6 Aggregation pipeline framework</td>
<td>19</td>
</tr>
<tr>
<td>3.4.7 MapReduce</td>
<td>19</td>
</tr>
<tr>
<td>3.5 Indexing support</td>
<td>20</td>
</tr>
<tr>
<td>Chapter 4 Cassandra qualitative evaluation</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Data model</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Architecture</td>
<td>23</td>
</tr>
<tr>
<td>4.2.1 Master-Master design</td>
<td>23</td>
</tr>
<tr>
<td>4.2.2 Availability, Consistency and Durability</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Data import</td>
<td>24</td>
</tr>
<tr>
<td>4.4 Query language</td>
<td>26</td>
</tr>
<tr>
<td>4.4.1 Create</td>
<td>26</td>
</tr>
<tr>
<td>4.4.2 Read</td>
<td>27</td>
</tr>
<tr>
<td>4.4.3 Update</td>
<td>27</td>
</tr>
<tr>
<td>4.4.4 Delete</td>
<td>28</td>
</tr>
<tr>
<td>4.4.5 Lightweight transaction support</td>
<td>28</td>
</tr>
<tr>
<td>4.4.6 Aggregate functions</td>
<td>28</td>
</tr>
<tr>
<td>4.4.7 User Defined Functions (UDF) and User Defined Aggregate functions (UDA)</td>
<td>28</td>
</tr>
<tr>
<td>4.5 Indexing support</td>
<td>29</td>
</tr>
<tr>
<td>4.6 Materialised views</td>
<td>30</td>
</tr>
<tr>
<td>Chapter 5 MongoDB and Cassandra quantitative evaluation</td>
<td>31</td>
</tr>
<tr>
<td>5.1 Dataset</td>
<td>31</td>
</tr>
</tbody>
</table>
5.2 Performance metrics .................................................................................................................32
5.3 Methodology ..............................................................................................................................32
  5.3.1 Data import workload .........................................................................................................32
  5.3.2 Read workload ....................................................................................................................32
  5.3.3 Write workload ..................................................................................................................33
5.4 Hardware and software specifications ......................................................................................33
  5.4.1 Hardware specifications ......................................................................................................33
  5.4.2 Software specifications ......................................................................................................33
5.5 Databases configuration ...........................................................................................................34
  5.5.1 Cassandra configuration ....................................................................................................34
  5.5.2 MongoDB configuration ...................................................................................................34
5.6 Performance results ..................................................................................................................35
  5.6.1 Data import workload .........................................................................................................35
  5.6.2 Read workload ....................................................................................................................38
  5.6.3 Write workload ..................................................................................................................40
Chapter 6 Amazon DynamoDB qualitative evaluation ..................................................................42
  6.1 Data model ..............................................................................................................................42
  6.2 Architecture ............................................................................................................................43
    6.2.1 Partitions and data distribution ........................................................................................43
    6.2.2 Availability, Consistency and Durability ..........................................................................44
  6.3 Data import ..............................................................................................................................44
  6.4 Query language .........................................................................................................................45
    6.4.1 Create ...............................................................................................................................45
    6.4.2 Read .................................................................................................................................45
    6.4.3 Update ..............................................................................................................................46
    6.4.4 Delete ...............................................................................................................................46
    6.4.5 Scan .................................................................................................................................47
    6.4.6 Query ...............................................................................................................................47
Appendix B  Examples of query language commands in MongoDB, Cassandra and Amazon DynamoDB ......................................................... 66

B.1  MongoDB ........................................................................................................ 66
B.1.1  Read ................................................................................................................ 66
B.2.2  Update .............................................................................................................. 66
B.2.3  Delete ................................................................................................................. 67
B.2.4  Text search ........................................................................................................ 67
B.2.5  Aggregation pipeline framework ....................................................................... 67
B.2.6  MapReduce ....................................................................................................... 68

B.2  Cassandra ............................................................................................................ 68
B.2.1  Read .................................................................................................................. 68
B.2.2  Update .............................................................................................................. 69
B.2.3  Delete ................................................................................................................. 69
B.2.4  Aggregate functions .......................................................................................... 69
B.2.5  User Defined Functions (UDF) and User Defined Aggregate functions (UDA) 69

B.3  Amazon DynamoDB ............................................................................................ 70
B.3.1  Create ................................................................................................................ 70
B.3.2  Read ................................................................................................................... 71
B.3.3  Update ............................................................................................................... 71
B.3.4  Delete ................................................................................................................ 71
B.3.5  Scan .................................................................................................................... 72
B.3.6  Query ............................................................................................................... 72

Bibliography .............................................................................................................. 73
List of Tables

Table 3.1 Tweet’s attributes .................................................................16
Table 5.1 Data set’s attributes ...............................................................31
Table 6.1 Amazon fine food reviews attributes .......................................45
Table 6.2 DynamoDB prices to achieve the minimum and maximum throughput achieved by MongoDB and Cassandra as presented in section 5 .........................49
Table 7.1 Summary table with the characteristics of the evaluated databases ..........52
List of Figures

Figure 2.1 Relational database data representation example ........................................5
Figure 2.2 Key-value model [16] .................................................................................. 7
Figure 2.3 An example of a document store document ................................................... 8
Figure 2.4 Wide-column store data model (Cassandra) [18] ........................................... 9
Figure 2.5 Graph store data model [20] ......................................................................... 9
Figure 3.1 Normalised representation of a blogging application (relational model)[26] ...................................................................................................................... 12
Figure 3.2 Denormalised representation of a blogging application (document model) [26] ...................................................................................................................... 12
Figure 3.3 Sharding in MongoDB [25] ............................................................................. 12
Figure 3.4 Query routing and sharding in MongoDB [26] .................................................. 13
Figure 3.5 Replica set in MongoDB [26] ........................................................................... 14
Figure 4.1 Cassandra column family .............................................................................. 22
Figure 4.2 Cassandra distributed design [32] .................................................................. 23
Figure 4.3 Cassandra can be deployed on hybrid on-premise/Cloud platforms [31] .... 24
Figure 5.1 Data import throughput ............................................................................. 35
Figure 5.2 Data import latency ...................................................................................... 37
Figure 5.3 Read throughput ............................................................................................ 38
Figure 5.4 Read latency .................................................................................................. 39
Figure 5.5 Write throughput ............................................................................................ 40
Figure 5.6 Write latency ................................................................................................ 41
Figure 6.1 DynamoDB data model .................................................................................. 42
Figure 6.2 Data distribution using the hash and range keys [40].................................43
Acknowledgements

First and foremost, I would like to thank my supervisor, Dr Charaka Palansuriya, for giving me the opportunity to focus and work on such an interesting topic and for his advice and support throughout the course of this dissertation project.

Additionally, I would like to thank EPCC’s System Development Team and especially Mr Donald Scobbie for preparing the hardware resources that I used and for their continuous technical support throughout the course of this project.

Finally, I would like to thank my parents for believing and trusting in me and for their encouragement and support throughout the whole year.
Chapter 1

Introduction

Since the invention of the first electronic general-purpose computer in 1946, ENIAC, technology advanced so fast that in the 1960’s the need for efficient storing, retrieval and processing of data led to the development of the first databases and Database Management Systems (DBMS) [1]. The first database systems that were developed in the 1960’s were navigational; they used pointers to allow navigation from a record to another. In 1970 Edgar F. Codd, an IBM researcher, developed the relational model and in the following years of the decade the first relational databases were developed based on it. They soon became very popular and they remain so until today. Relational databases were the favourable means of data storage for many companies and institutions for many years, but since the late 2000’s, NoSQL databases have been gaining popularity [2]. NoSQL databases were developed to efficiently store and manage big data; they have a distributed design and they aim to provide horizontal scalability, high availability and schema flexibility. Some other types of databases such as object-oriented and RDF databases were developed through the years and they still do exist, but they are not very popular.

According to db-engines.com, a website that ranks database systems according to their popularity, the most popular DBMS today are the relational and NoSQL ones; in the 50 most popular systems, there are 28 relational databases and 22 NoSQL databases [3]. Relational systems top the popularity rankings due to the fact that through the years this type of databases proved to be very reliable and efficient. However, they are based on a model that was developed 46 years ago and today they face some limitations. These limitations arose in the recent past when rapid technology development and the web 2.0 led to a data deluge. Today, many institutions and companies produce data in high volumes, with high velocity and in a great variety of forms that relational databases cannot efficiently handle [4]. For this reason, NoSQL databases emerged to provide an alternative solution that is scalable, distributed, more flexible and less costly [4].

1.1 Dissertation purpose

There is a wide variety of available NoSQL databases today for a user to choose from and each one focuses on a specific area, aiming to solve a particular problem; not
every NoSQL database is suitable for every use case. The main goal of this dissertation project is the evaluation of three major NoSQL databases; MongoDB, Cassandra and Amazon DynamoDB. MongoDB and Cassandra will be both qualitatively and quantitatively evaluated and compared with each other, with the ultimate goal of investigating their main features, strengths and weaknesses and identifying and recommending suitable use cases for each database. The qualitative evaluation will include the following quality attributes:

- Availability
- Scalability
- Durability
- Consistency
- Supported data models and suitable types of data
- Schema flexibility
- Expressive power of the supported query language
- Indexing options and support
- Ease of use

The qualitative evaluation will be followed by a quantitative evaluation, during which the following metrics will be used:

- **Data import performance**: Latency (average time for one operation) and throughput (operations/second) achieved when importing the same set of data into each database

- **Read performance**: Latency and throughput achieved when executing the same set of read operations in each database

- **Write performance**: Latency and throughput achieved when executing the same set of update operations in each database

Amazon DynamoDB will be qualitatively evaluated using the same quality attributes that will be used for the other two databases. However, a quantitative evaluation will not follow as DynamoDB is a Database as a Service (DBaaS) that is hosted within Amazon’s public cloud, and therefore, a performance comparison between databases deployed on different hardware resources would not be objective. Instead, an evaluation of using a DBaaS will be performed in order to identify the benefits and drawbacks of using a public cloud-based solution. The additional quality attributes that will be used for the evaluation of the public cloud-based solution will be:

- **Cost**: Investigate costs of using DBaaS instances for use cases with varying requirements

- **Response time**: Measure the response time associated with Create Read Update Delete (CRUD) operations
- **Elasticity and on-demand self-service**: Investigate the ease of scaling the resource usage up or down without human interference

- **Ease of use**: Investigate how easy it is to configure and use a DBaaS instead of a manually deployed database.
Chapter 2

Background Theory

In this chapter, some data challenges that modern technology brought are presented. Consequently, an overview of the relational databases is presented, stating their structure, properties, benefits and limitations, followed by an overview of the four types of NoSQL databases. Finally, some related work that evaluated NoSQL databases is presented.

2.1 Big Data

In the late 2000’s, rapid technology development and the web 2.0 led to a data deluge. Data generation rates increased exponentially; according to IBM, 90% of the data in the world today has been created in the last two years alone [10]. YouTube users upload 48 hours of video every minute; Facebook users upload 100TB of data every day; Walmart handles more than 1 million customer transactions every hour [11]. Moreover, data today are being generated in many different kinds and forms; structured, semi-structured and unstructured text, images, videos and audio data are just some examples. All these led to the coining of the term “Big Data” which can be described by the following five Vs [12]:

- **Volume:** The massive amount of stored data
- **Variety:** The various different kinds and forms of data
- **Velocity:** The speed at which data is generated and can be processed
- **Veracity:** The quality and accuracy of the data
- **Value:** The actual value we can get from the data

There are additional Vs that are occasionally used to describe big data as well; Validity, Variability, Venue, Vocabulary and Vagueness [13]. However, database systems are primarily concerned with the first three, Volume, Variety and Velocity.
2.2 Relational databases

In 1970, Edgar F. Codd, an IBM researcher, published a paper [5] describing how data could be represented as a collection of relations. Towards the end of the decade, the first database management systems built after the relational model were developed and they soon became very popular. In relational databases, data is represented as a collection of tables (relations). Each row of a table represents an entity, and each column of a table represents an attribute of the entities. Each entity is uniquely identified by a set of attributes, known as the primary key. Two or more tables can be related with one another via foreign keys, fields of one or more columns in a table that reference the fields of the primary key of another table. Database operations (e.g., table creation/deletion, CRUD operations) are expressed using the Structured Query Language (SQL), a powerful and very expressive declarative programming language designed specifically for data management.

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Name</th>
<th>Surname</th>
<th>Date of Birth</th>
<th>Store ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>573237</td>
<td>John</td>
<td>Smith</td>
<td>12/08/1979</td>
<td>1</td>
</tr>
<tr>
<td>984318</td>
<td>George</td>
<td>Anderson</td>
<td>03/05/1988</td>
<td>2</td>
</tr>
<tr>
<td>194372</td>
<td>Julia</td>
<td>Jacobs</td>
<td>21/11/1985</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store ID</th>
<th>City</th>
<th>Phone number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>London</td>
<td>020 723 3456</td>
</tr>
<tr>
<td>2</td>
<td>Edinburgh</td>
<td>013 149 0018</td>
</tr>
</tbody>
</table>

Figure 2.1 Relational database data representation example

In figure 2.1 an example of a data set stored in a relational database is presented. The dataset consists of two tables; the table Employee, which stores information about the employees of a company and the table Store, which stores information about the individual stores of the company in the country. The primary keys are Employee ID and Store ID, and the uniquely identify the entities of Employee and Store respectively. The attribute Store ID of the table Employee is a foreign key, as it references the primary key of the table Store.

2.2.1 Database normalisation

The concept of normalisation was introduced by Codd in 1970, and it aims to organise columns and tables of a relational database in a manner that minimises data redundancy [6]. There are three basic normal forms [45]:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
• **First Normal Form (1NF):** A table for each set of related data is created, each set of related data is identified by a primary key, and repeating sets of related data in an individual table are eliminated.

• **Second Normal Form (2NF):** Every non-prime attribute of each table must depend on the entire candidate key, not part of the candidate key.

• **Third Normal Form (3NF):** There are no non-prime attributes that depend on another non-prime attribute, non-prime columns must depend only on the primary key.

In order to satisfy a normal form, a table must satisfy all of its predecessor forms. While higher forms exist (BCNF, 4NF, 5NF, 6NF), they are rarely used in practice and a database table is considered normalised if it satisfies the 3NF.

2.2.2 ACID transactions

One important benefit that relational database provide is the support of ACID transactions. A transaction is a sequence of database operations that represents a single unit of work. The ACID properties ensure that each transaction keeps the database in a valid, consistent and reliable state by guaranteeing the following [46]:

• **Atomicity:** If the transaction involves more than one action, either all of them are executed or none at all.

• **Consistency:** A transaction brings the database from one valid state to another.

• **Isolation:** In concurrent execution of transactions each transaction is guaranteed to produce results that would have been produced if the execution was serial.

• **Durability:** Once committed, a transaction is guaranteed to be stored permanently in non-volatile memory.

2.2.3 Limitations

Relational databases became a commercial success soon after the initial prototypes were released and they remain so until today. The main reasons are the support of ACID transactions that guarantee reliable data processing which is very important in many business areas (e.g., financial transactions), and the support of SQL which is very powerful and expressive and provides easy and efficient data management. However, being based on a model that is 46 years old, relational databases seem to have some limitations today. They traditionally relied on vertical scalability, that is increasing the hardware resources of a single node instead of adding more less powerful nodes and building a cluster; that scan prove to be rather expensive and impractical [7, 8]. Some relational databases evolved to support a distributed design
to scale horizontally over multiple nodes (e.g. MySQL Cluster) but their performance suffers when the volume of data is increased; operations (e.g., join multiple tables) and transactions that span many nodes induce additional communication and two-phase commit overheads [9]. Finally, relational databases require the definition of a fixed, tabular schema before importing data and that means that the data must be able to be stored into tables.

2.3 NoSQL Databases

The data generation trends described in 2.1 and the limitations of the relational databases led to the emergence of a new type of databases, the NoSQL databases, which aim to provide novel mechanisms to efficiently store and manage big data. NoSQL means Not-only-SQL and the term includes a set of databases that do not use SQL as a query language, they support distributed designs, flexible schemas, semi-structured and unstructured data, they do not support complex relationships and they provide high availability and horizontal scalability [4]. In order to provide high performance, availability and scalability, NoSQL databases do not provide ACID properties to the level that the relational databases do but instead, they follow the BASE model [14]:

- **Basically Available**: The system provides high availability
- **Soft state**: The state of the system can change over time, even if there is no input due to the eventual consistency model
- **Eventual consistency**: The system will eventually become consistent at some point later in time

There are four main types of NoSQL databases, each one adopting a different data model and following a different design: key-value stores, document stores, wide column stores and graph stores.

2.3.1 Key-value stores

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA,BBB,CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA,BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA,DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA,2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3,ZZZ,5623</td>
</tr>
</tbody>
</table>

Figure 2.2 Key-value model [16]
The key-value model is the simplest type of NoSQL databases. As it is shown in figure 2.2, data is stored in the form of an associative array of key-value pairs. Keys are usually strings and each key is associated only with one value. Values can be any type of data like text, images or documents, and in pure key-value models, values are opaque to the database. For this reason, in general, these stores support simple get, put and delete commands instead of a more complex query language. Due to their simple design, key-value stores are easy to use and they provide high performance, scalability and flexibility [15]. Some examples of key-value stores are Redis, Memcached and Amazon DynamoDB [3].

2.3.2 Document stores

Document stores are databases that store and manage semi-structured, document-oriented data (usually JSON or XML documents). The documents consist of key-value pairs, where keys represent the attributes of a document and the values represent the actual values of the attributes. Each document is uniquely identified by a single attribute, the primary key. In figure 2.3 an example of a JSON document that can be stored in a document store can be seen. The attributes of the document can be seen on the left side of the colon and the values on the right side. The primary key of the document is the “_id” attribute with the value “5435dc4327a43”. Document stores allow complex queries to be performed based on the data and the attributes of a document. Finally, document stores are schema-free; the user does not have to declare a schema before importing data into the database [17]. Some examples of document stores are MongoDB, CouchDB and CouchBase [3].
2.3.3 Wide-column stores

<table>
<thead>
<tr>
<th>Row ID</th>
<th>Name</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preston</td>
<td><a href="http://www.example.com">www.example.com</a></td>
</tr>
<tr>
<td>2</td>
<td>Julia</td>
<td><a href="http://www.example.com">www.example.com</a></td>
</tr>
<tr>
<td>3</td>
<td>Alice</td>
<td><a href="mailto:example@example.com">example@example.com</a> <a href="http://www.example.com">www.example.com</a></td>
</tr>
</tbody>
</table>

![Figure 2.4 Wide-column store data model (Cassandra) [18]](image)

Wide-column stores can be described as two-dimensional key-value stores that support large numbers of columns. As it can be seen in figure 2.4, a row in a wide-column store consists of a collection of column name-value pairs, and a column family (the equivalent of a relational table) consists of a collection of similar rows. Multiple rows in the same column family do not have to contain the same number of columns. Wide column stores usually require a schema to be defined up-front, but this can be altered on-the-fly with little to almost no cost at all [19]. The query language capabilities of this type of databases depend on the specific implementation of this model. Some examples of wide columns stores are Cassandra, HBase and Accumulo [3].

2.3.4 Graph stores

![Figure 2.5 Graph store data model [20]](image)

Graph stores are databases that use graph structures to manage, store and retrieve data. As it can be seen in figure 2.5, in graph stores data is represented as entities (nodes) and relationships (edges) that act as connections between entities. Both the entities and the relationships can store a number of attributes in the form of key-value pairs. These
types of databases are suitable for complex queries on highly connected data [21]. Some examples of graph stores are Neo4J, OrientDB and Titan [3].

2.4 Related work

2.4.1 Quantitative evaluations

The first NoSQL databases were developed about a decade ago, but there are more than 225 systems available today [22]. As the number of available systems increased, various researchers and companies became interested in evaluating the different technologies and comparing them with each other, using benchmarks to evaluate their performance.

In 2015, End Point, a database and open source consulting company, evaluated the performance of MongoDB, Cassandra and other NoSQL databases [23]. End Point used the Yahoo! Cloud Serving Benchmark (YCSB) on Amazon Web Services (AWS) EC2 instances. They configured the databases to have durable writes, and they used data volumes that exceeded the RAM capacity for each node, so as to represent ‘big data’. They evaluated the performance of the databases (operations/second) using six different workloads, on 1, 2, 4, 8, 16 and 32 nodes. Cassandra outperformed MongoDB in all six workloads:

- **Data import**: Cassandra had up to 2.9X higher throughput
- **Read-mostly workload**: Cassandra had up to 39X higher throughput
- **Balanced Read/Write mix workload**: Cassandra had up to 140X higher throughput
- **Read-Modify-Write workload**: Cassandra had up to 105X higher throughput
- **Mixed Operational and Analytical workload**: Cassandra had up to 568X higher throughput
- **Insert-mostly workload**: Cassandra had up to 24X higher throughput

End Point’s evaluation results indicate Cassandra as a clear performance winner. However, in another evaluation in 2015, independent evaluators United Software Associates showed that MongoDB outperforms Cassandra [24]. These evaluators used three different durability configurations and two different workloads to benchmark the databases on a single database server. The performance results (operations/second) can be seen below:

**Workload A (50% Read, 50% Update):**

- **Throughput optimised**: MongoDB had 1.19X higher throughput
- **Durability optimised**: MongoDB had 5.1X higher throughput
- **Balanced**: MongoDB had 1.47X higher throughput

**Workload B (95% Read, 5% Update)**

- **Throughput optimised**: MongoDB had 1.36X higher throughput
- **Durability optimised**: MongoDB had 2.1X higher throughput
- **Balanced**: MongoDB had 2.56X higher throughput

These two evaluations demonstrate that a performance evaluation of MongoDB and Cassandra is dependent on various configuration parameters, different types of workloads and different numbers of nodes. Additionally, both evaluations used workloads that combined insert/read/update operations together; they did not assess the performance of each type of operations alone (e.g., read performance, insert performance).

### 2.4.2 Qualitative evaluations

Database performance is an important factor when deciding which database system to use, but there are some other features to consider as well. The data model, schema flexibility, query language expressiveness, indexing support, and ease of use of a database system may be more important than the performance itself for many use cases. In [34] and in [35], a qualitative comparison between MongoDB, Cassandra and other NoSQL systems is presented by stating the characteristics as well as the advantages and disadvantages of each system.

Cassandra provides high availability and scalability, and by default eventual consistency. The consistency can be configured to be strong at the expense of additional latency. It can be easily set up and maintained, and in general, it has low cost of ownership. Cassandra has a decentralised, master-master architecture with no single point of failure, where every node can perform any operation. It supports secondary indexes and fast reads and writes. However, it has a relatively weak query language that does not have many capabilities, it does not support dynamic, ad-hoc queries neither transactions and it is not suitable for relational data.

MongoDB provides high availability and scalability via auto-sharding and master-slave replication, as well as strong consistency by default (it is configurable). It is completely schema-free, as no data structures or database schema has to be defined up-front. MongoDB provides a rich and powerful query language that support complex, ad-hoc queries. Furthermore, it provides full index support to enhance query performance. However, like Cassandra, it does not support transactions.

To conclude, most evaluations of NoSQL databases follow either quantitative approach, focusing on a performance evaluation or a qualitative approach focusing on the features of each database system. This dissertation project will follow both a qualitative and quantitative approach. The qualitative and quantitative features stated in section 1.1 will be evaluated, with the aim of obtaining an overall understanding of the strengths and weaknesses of the databases and of identifying and recommending suitable use cases for each database.
Chapter 3

MongoDB qualitative evaluation

MongoDB is an open-source, NoSQL document-oriented database that stores data in the form of JSON-like documents. Initially released in 2009, MongoDB is currently the most popular document store and the fourth most popular database management system.

3.1 Data model

MongoDB stores data in BSON format, a binary representation of JSON documents. BSON documents encode information as an ordered list of key-value pairs, where keys are string and values can be string, integer, double, date, Boolean, byte array, BSON array, BSON object (sub-document) or null.

Figure 3.1 Normalised representation of a blogging application (relational model)[26]

Figure 3.2 Denormalised representation of a blogging application (document model) [26]
Documents in MongoDB are organised into collections, as records are organised into tables in relational databases. However, the main difference between MongoDB and relational databases is that MongoDB is schema-free. The structure of the documents within a collection may be different, no schema has to be declared, documents do not need to share the same attributes, and additional fields can be added to a document without affecting any other document of the collection.

Another key difference is that the data in the relational model is normalised. That is, related data are split and organised into separate tables with the aim of minimising data redundancy. In figure 3.1 an example of the normalised model for a blogging application can be seen: Information about an article is separated in five different relations (Article, User, Category, Tag and Comment). If the user wants to retrieve the comments or the tags of an article for example, expensive join operations must be performed in order to bind the data together.

On the other hand, in MongoDB’s document model data are denormalised and stored in just two separate collections, User and Article. As illustrated in figure 3.2, multiple comments, tags and categories can be added in the same document as they are stored as arrays. This results in a higher performance and scalability than relational databases, as the need of join operations is completely eliminated and all the data can be retrieved in a single read operation [26].

3.2 Architecture

3.2.1 Auto-sharding

Mongo provides horizontal scalability on cluster or cloud infrastructures using a technique called auto-sharding. This technique divides and distributes the data set over multiple physical partitions (shards). In figure 3.3 an example of sharding in MongoDB can be seen: A 1 TB dataset is divided into four smaller parts of 256 GB each and distributed to different physical partitions. Each partition handles independently its own data, and all the partitions together make up a single logical database.

![Figure 3.3 Sharding in MongoDB](image-url)
MongoDB distributes the data of a single collection based on the shard key, an indexed field that exists in every document of the collection. Currently, three sharding policies are supported; range-based, hash-based and location-aware sharding. In range-based sharding, documents that their shard key is close to each other are likely to be placed in the same shard; in hash based-sharding documents are uniformly distributed across all shards; in location-aware sharding, documents are distributed according to user configurations that associate shard key ranges with specific shards and physical resources.

The sharding technique is transparent to applications and is built-in to the system; MongoDB automatically balances shards across physical partitions as data or physical resources increase or decrease. As illustrated in figure 3.4, application issued queries are handled by a query router. Based on the query type, the query router will either dispatch it to the shard or shards that contain the documents that have to be returned, or it will broadcast it to every shard in the database. Multiple query routers can coexist to provide additional performance and availability [25, 26].

![Query routing and sharding in MongoDB](image)

**Figure 3.4 Query routing and sharding in MongoDB [26]**

### 3.2.2 Availability, Consistency and Durability

MongoDB ensures high availability by maintaining multiple copies of shards, called replica sets. A replica set consists of multiple replicas (copies of a shard) and each replica can be stored on a different machine, rack, or even data centre to maximise data safety. In single server deployments, MongoDB can use the journal to ensure data durability, a mechanism similar to the transaction log that the relational databases use. The journal keeps an ordered set of operations that took place in the database, and it can be configured to be flushed to disk every time a new operation is appended to it. This way, even after a power failure the journal can be replayed and any lost operation can be executed again.
As illustrated in figure 3.5, all replica sets have a primary replica which serves all reads and writes by default to ensure strong consistency. However, in scenarios where slightly outdated data are acceptable, reads can be performed from secondary replicas, which are eventually consistent with the primary one. In regular time intervals, secondary replicas get updated using the oplog, a log file that keeps an ordered set of operations that took place on the primary replica. Finally, when the primary replica fails, a secondary one assumes its role and becomes itself primary [26].

Although MongoDB does not support transactions, it is considered as ACID compliant at the document level. Multiple fields of a single document can be updated in a single write operation. However, if any errors occur during the update, the operation will be rolled back and the database will remain in a consistent state. Furthermore, MongoDB allows the user to specify various policies regarding the commitment of the operation that range from unacknowledged (no proof that the operation was committed), to fully acknowledged that it has been committed across all replicas of the shard [26].

### 3.3 Data import

In order to evaluate the features of MongoDB, a data set had to be imported. As MongoDB is a document-oriented database that stores data as JSON-like documents, data sets in JSON format are good candidates to be imported as much less pre-processing is required. The data set that was selected to be imported was twitter’s user tweets, as provided by the twitter streaming API.

The streaming API is a service provided by twitter that allows the retrieval of public data (tweets) as they flow through twitter over a persistent HTTP connection. Twitter offers a REST API as well; however, it limits the number of requests a user can do in a specific amount of time. The streaming API however, can provide the developers with a theoretically infinite amount of data as the HTTP connection that streams tweets remains active until the client closes it. The tweets are retrieved as a continuous flow of JSON objects, with each JSON object consisting of many attributes such as id, tweet text, retweet count and user information. In table 3.1 a complete list of attributes that
are present in a tweet’s JSON document is presented. The twitter data set was selected to be imported to the database because it is in JSON format, it has a fairly complex structure including array attributes and sub-documents (for example, the “entities” attribute is a sub-document that contains three arrays of additional nested sub-documents), and because an arbitrary number of tweets can be retrieved using the streaming API.

The tweets were retrieved as a single, flowing stream of data and they were stored in a temporary file. Consequently, a python script was implemented to parse the file, select each individual tweet and import it to MongoDB. Neither a database schema nor a collection had to be defined up-front; both were created automatically when the first tweet got imported. Approximately 1 million tweets (about 2 GB of data) were imported.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotations</td>
<td>Object</td>
</tr>
<tr>
<td>contributors</td>
<td>Array of objects</td>
</tr>
<tr>
<td>coordinates</td>
<td>Object</td>
</tr>
<tr>
<td>created_at</td>
<td>String</td>
</tr>
<tr>
<td>current_user_retweet</td>
<td>Object</td>
</tr>
<tr>
<td>entities</td>
<td>Object</td>
</tr>
<tr>
<td>favourite_count</td>
<td>Int</td>
</tr>
<tr>
<td>favourite</td>
<td>Boolean</td>
</tr>
<tr>
<td>filter_level</td>
<td>String</td>
</tr>
<tr>
<td>geo</td>
<td>Object</td>
</tr>
<tr>
<td>id</td>
<td>Int64</td>
</tr>
<tr>
<td>id_str</td>
<td>String</td>
</tr>
<tr>
<td>in_reply_to_screen_name</td>
<td>String</td>
</tr>
<tr>
<td>in_reply_to_status_id</td>
<td>Int64</td>
</tr>
<tr>
<td>in_reply_to_status_id_str</td>
<td>String</td>
</tr>
<tr>
<td>in_reply_to_user_id</td>
<td>Int64</td>
</tr>
<tr>
<td>in_reply_to_user_id_str</td>
<td>String</td>
</tr>
<tr>
<td>lang</td>
<td>String</td>
</tr>
</tbody>
</table>
Database developers can interact with MongoDB either by using the mongo shell, which is by default installed with MongoDB, or by using a MongoDB driver for the chosen programming language. The information presented in this section and in section 3.5 is described in [37]. Examples of the commands presented in this section can be seen in appendix B.

### 3.4 Query language

MongoDB provides three methods to insert documents into a collection:

- `db.collection.insertOne()`: Inserts one document to the collection
- `db.collection.insertMany()`: Inserts many documents to the collection
- `db.collection.insert()`: Inserts either one or many documents to the collection
When a single document needs to be imported into the database, it is specified between the parentheses of the insertOne() or insert() commands. In cases where multiple documents need to be imported into the database together (bulk writes), an array of documents is specified between the parentheses of the insertMany() or insert() commands. During the import procedure, the data from twitter were parsed and each tweet got imported individually into the database using the following script:

```python
with open(str(fileName)) as file:
    for line in file:
        db.tweets.insert_one(json.loads(line))
```

Each tweet was on a separate line, so in the above command the contents of each tweet (line) were encoded to JSON, and then imported into the tweets collection, one by one.

If a collection does not exist when an insert command is executed, the collection is automatically created. Additionally, all the documents stored in a collection require a unique “_id” field to serve as the primary key, and if it is not provided, it is automatically created by the system. Finally, write operations in MongoDB are atomic to the document level. That is, either all the fields of a document will be written to the database and stored to disk, or none.

### 3.4.2 Read

MongoDB provides a simple command for basic read operations:

```mongodb
db.collection.find(<query filter>, <projection>)
```

In the `<query filter>` clause, conditions can be specified so as to filter the results, and in the `<projection>` clause, the fields which will be projected can be specified as well. Both clauses are optional, and if they are not specified in the command, the database will return all the fields of every single document in the collection.

MongoDB allows the use of multiple operators together in the `<query filter>` clause, like comparison, logical, evaluation, geospatial, bitwise and array operators. Additionally, it allows queries on embedded documents and on array fields.

### 3.4.3 Update

MongoDB provides three methods to update documents of a collection:

- `db.collection.updateOne(<query filter>, <fields to update>):` Updates a single document of a collection
- `db.collection.updateMany(<query filter>, <fields to update>):` Updates multiple documents of a collection
- `db.collection.update(<query filter>, <fields to update>):` Updates a single document or multiple documents of a collection
- `db.collection.replaceOne(<query filter>, <new fields>):` Replaces a single document of a collection
In the <query filter> clause, conditions can be specified so as to determine which documents have to be updated. MongoDB allows the use of multiple query operators together in the query filter clause, as explained in section 3.4.2. In cases where only a single document has to be updated or replaced, the updateOne()/replaceOne() command is used, and in cases where multiple queries need to be updated together, the updateMany() command is used. In the <fields to update> clause, the fields that need to be updated are specified, along with the update operator.

### 3.4.4 Delete

MongoDB provides three methods to delete documents from a collection:

- `db.collection.deleteOne(<query filter>):` Deletes one document from the collection
- `db.collection.deleteMany(<query filter>):` Deletes many documents from the collection
- `db.collection.remove(<query filter>):` Deletes either one or many documents from the collection

In the <query filter> clause, conditions can be specified so as to determine which documents have to be deleted. MongoDB allows the use of multiple query operators together in the query filter clause for this operation as well, as explained in section 3.4.2. If nothing is specified in the <query filter> clause of the “deleteMany” or “remove” command, then all the documents of the collection are deleted.

### 3.4.5 Text search

MongoDB supports advanced text search query operations on fields of string content or on arrays of strings. However, in order to enable this feature, a text index has to be created on the corresponding string field (more information about MongoDB indexes is presented in section 3.5). The text index can be created using the following command:

```javascript
db.collection.createIndex( { <String field name>: "text"} )
```

Once the index is created, text search operations can be executed on the indexed field by using the “$text” and “$search” operators in the “find()” command. The “$text” operator uses whitespaces and most punctuation as delimiters to tokenize the search string specified in the “$search” operator. Consequently, the query is executed and the database returns any document of the collection that contains at least one of the tokenized words. The “$text” operator will not tokenize words that are wrapped in double-quotes so as to allow exact phrase search. Additionally, it is allowed to exclude documents that contain specific words from the results by using the “-” operator before those words.
3.4.6 Aggregation pipeline framework

MongoDB provides the aggregation pipeline framework that allows the implementation of complex, multistage queries that transform the documents into aggregate results. The basic syntax for aggregation pipeline queries is the following:

```javascript
db.collection.aggregate([ { <Operation 1> },...., { <Operation N> } ])
```

The aggregation pipeline commands work in a similar way to the UNIX shell pipelined commands; the execution consists of multiple stages, and each stage receives an input, applies a transformation to it and forwards the output to the next stage. The aggregation pipeline framework provides a rich set of operators that can be applied to each stage; some examples include specific field projection, group by a specific field, sort results, limit result set and even a left outer join operator. This framework is a very powerful and useful tool that enhances the expressiveness of MongoDB’s query language; using its rich set of operators, users can transform most SQL commands (except inner, full, cross and right outer joins) into MongoDB commands and perform complex queries. However, in cluster deployments of MongoDB, if the aggregation pipeline framework is used the performance will degrade significantly (compared to the performance that MongoDB achieves when simple “find” commands are used instead). The queries will have to be broadcasted to every node in the cluster and the results between the pipeline stages will have to be gathered at a single node; subsequently, the results will have to be scattered back to every node again so as to continue with the execution of the next stage in the pipeline.

3.4.7 MapReduce

As an alternative to the aggregation pipeline framework, MongoDB provides also the MapReduce framework to support aggregation operations. This framework follows the classic MapReduce paradigm; the map function scans all the documents of a collection, does some processing and then outputs key-value pairs, while the reduce function takes the output of the map function and performs aggregation operations. The basic syntax for MapReduce queries is the following:

```javascript
db.collection.mapReduce(
  function map(){emit ( key, value ); },
  function reduce( key, value ){ return value; },
  {
    query: { <query operators>},
    out: <collection name>
  }
)
```

Both the map and reduce functions are written in JavaScript. The results are by default returned as a document; however, if a collection name is specified in the “out” clause of the command, they are stored permanently in that collection. Finally, in the “query” clause, the user can specify some query criteria so that the operation will be performed only on those documents of the collection which meet the specified criteria. Similarly to the aggregation pipeline framework, the MapReduce framework is a powerful tool...
that enhances MongoDB’s query language. It can prove to be very useful to users who have experience in using the MapReduce programming paradigm. However, in cluster deployments of MongoDB it can cause the performance to degrade as well (compared to the performance that MongoDB achieves when simple “find” commands are used instead). The map functions will have to be executed on every node in the cluster, the map outputs will have to be sent to the reducers and finally the reduce outputs will have to be gathered together and sent back to the user.

3.5 Indexing support

By default, MongoDB creates a primary index on the _id attribute of each document during the creation of a new collection. This index is unique and permanent; that is, it does not allow duplicated _id values in the same collection, and it cannot be dropped. However, in order to support efficient query execution, MongoDB allows the users to create secondary indexes on any other document attribute as well. The basic command to create a secondary index is the following:

```
   db.collection.createIndex( <indexed key and index type>, <options> )
```

In the indexed key/type specification, the user can specify which field or combination of fields will be indexed, and what type the index will be of. The types of supported indexes are the following:

- **Single field**: Ascending/Descending index on a single field
- **Compound**: Ascending/Descending index on a combination of fields
- **Multi-key**: Ascending/Descending index on the values of an array field
- **Text**: Supports text search on a string field
- **Geospatial**: Supports efficient queries of geospatial data
- **Hashed**: Indexes the hash of the value of a field – It does not support for range queries

In the <options> clause, the user can specify additional properties for an index:

- **Unique**: Rejects duplicate values for the indexed field
- **Partial**: Indexes only the documents of the collection that meet a specific filter expression
- **Sparse**: Indexes only the documents of the collection that contain the indexed field
- **Time To Live (TTL)**: Automatically deletes the indexed documents from the collection after a period of time

In general, indexes in MongoDB follow the same logic that indexes in relational databases follow. They are B-tree data structures that store the values of the indexed field in an ordered manner. Therefore, a general suggestion is to create secondary indexes on the fields of a collection that are used in query clauses so as to improve the query performance.
Chapter 4

Cassandra qualitative evaluation

Cassandra is an open-source, NoSQL wide column database that was initially developed at Facebook by Avinash Lakshman (who had also worked on the Amazon Dynamo system) and Prashant Malik. In 2009 Cassandra became an Apache project and is currently the most popular wide column store and the seventh most popular database management system.

4.1 Data model

At first glance, Cassandra’s data model looks similar to the relational one. Cassandra organises data in column families (or tables) like the relational databases organise data in tables. Cassandra is not schema-free; column families have to be declared upfront, by stating the name of the column family/table and by declaring the name and the type of the columns, as well as defining a primary key, using Data Description Language (DDL) commands. Each row both in the relational model and in Cassandra can be considered as an individual record [29]. However, there are a lot of major differences between the two.

Cassandra is a partitioned row store and that makes the choice of the primary key very important. Apart from uniquely identifying a row, in Cassandra the primary key also defines the way and the order that the data is stored. Cassandra primary keys are usually compound (consist of more than one attribute) and their first component is called the partition key. Rows that have the same partition key are grouped and stored together in the same partition. Within a partition, the rows are sorted by the remaining columns (clustering columns) of the primary key [27, 28].
In the relational model, each column has only a value. The name of the column is stored as meta-data, during the definition of the schema. In Cassandra each column is stored as a combination of key-value pairs, where the key is the name of the column and value is the actual value. Another difference is that, while in the relational model each row in a table has to have exactly the same amount of columns, in Cassandra it does not. Different rows may have a different number of columns. This feature allows a more flexible schema. Columns in Cassandra can be added on-the-fly as needed; previously inserted rows will remain unaffected, and rows that will be inserted in the future may use any number of columns they need [29]. An example of a column family can be seen in figure 4.1. The partition key of this column family is the column “City” and therefore, rows that have the same value for this column are stored together on the same partition; within a partition rows are sorted by the column “Surname”, which is the clustering key.

The table design in Cassandra uses a query driven, denormalised approach that contrasts with a relational database approach that is table driven and normalised. Due to the fact that Cassandra does not support join operations, queries are designed to access only a single table. Furthermore, Datastax, the company that distributes Cassandra, suggests that a good practice is to maintain a different table for each individual query (more details in section 4.4.2). Therefore, it is not uncommon in Cassandra to have replicated data across many different tables [27].
4.2 Architecture

4.2.1 Master-Master design
Cassandra did not adopt the classic master-slave or the sharded design. Instead, it uses a peer-to-peer distributed architecture, following the master-master design where all nodes are the same and they exchange their state information using the gossip protocol. Data are automatically distributed in Cassandra across all nodes that participate in a cluster. The partition process is completely transparent to the user; the only thing they have to choose is the partition strategy. As illustrated in figure 4.2, each node in a Cassandra cluster is assigned a range of token ranges. When a new row is inserted, the token for that row is computed by using its partition key as input to a hash function, and then this token is used to determine the node to which the row will be stored.

When the user needs to store or retrieve data in Cassandra, they can connect to any node in the cluster. This node then serves as the coordinator, and it can forward or request data to and from the nodes that hold it. This distributed architecture allows Cassandra to scale horizontally and handle Petabytes of data and thousands of concurrent operations per second, across multiple data centres with ease [31].

![Figure 4.2 Cassandra distributed design][32]

4.2.2 Availability, Consistency and Durability
Cassandra provides built-in, automatic replication. Data can be replicated many times on different racks, data centres and cloud platforms, as Cassandra clusters can be deployed on hybrid on-premise/could platforms (figure 4.3). Cassandra exploits this distributed design and replication feature to provide true continuous availability, as there is no single point of failure. When a node or rack fail, the database can still work
without going down by using replicated data. The replication strategy as well the replication factor is defined for each individual database when they are created.

Cassandra can be deployed on hybrid on-premise/Cloud platforms [31]

Cassandra allows the users to set the level of consistency they wish. It can scale from eventual consistency (only one node responds positively, and the others get updated eventually) to strong consistency (every node responds positively). Additionally, different levels of consistency can be set for different operations (SELECT, INSERT, UPDATE, DELETE). Strong consistency ensures that data are consistent at all times across all the nodes in the cluster. However, eventual consistency can give a great boost to performance as the need for constant synchronisation between the nodes is eliminated.

Cassandra uses the commit log to ensure durability, a mechanism similar to the journal that MongoDB uses. When a write occurs, Cassandra stores the data in main memory using a structure called memtable. Memtables are stored to disk when their size exceeds a threshold. To ensure durability, every write is appended to the commit log and consequently the commit log is flushed to disk. This way, even after a power failure the data is guaranteed to be stored permanently. Durable writes and the use of the commit log can be set to on or off during the creation of a new database [31].

4.3 Data import

In order to evaluate the features of Cassandra, a data set had to be imported. Cassandra is a wide column store that stores data in column families, similarly to the way that relational databases store data in tables. Therefore, data sets in CSV format are good candidates to be imported as less pre-processing is required. The data set that was selected to be imported was Expedia’s customer behaviour data set that was provided for Kaggle’s Expedia Hotel Recommendation competition [33].
This data set consists of a random selection of the online behaviour of Expedia’s customers. The attributes of this data set include mainly customer search information, like what did the customers search for, in which countries they were interested in and whether they made a booking or not. The full list of attributes can be seen in the Cassandra Query Language (CQL – Cassandra’s equivalent to SQL) command below. The Expedia customer behaviour data set was selected to be imported to the database because it is in CSV format, it is fairly large (approximately 3.8 GB) and it is publicly available on Kaggle.

Before importing the data into Cassandra, a Keyspace (Cassandra’s equivalent to a relational database’s “database”) and a column family had to be defined. As the qualitative evaluation would take place on a single node deployment, the replication factor for the Keyspace was set to one (no replication) and the durable writes option was set to true. Consequently, a column family was created using the following CQL Data Description Language (DDL) command:

```cql
CREATE TABLE SEARCH_DATA(
    DATE_TIME TEXT,
    SITE_NAME INT,
    POSA_CONTINENT INT,
    USER_LOCATION_CONTINENT INT,
    USER_LOCATION_COUNTRY INT,
    USER_LOCATION_REGION INT,
    USER_LOCATION_CITY INT,
    ORIG_DEST_DISTANCE DOUBLE,
    USER_ID INT,
    IS_MOBILE INT,
    IS_PACKAGE INT,
    CHANNEL INT,
    CHECKIN_DATE TEXT,
    CHECKOUT_DATE TEXT,
    ADULTS INT,
    CHILDREN INT,
    ROOMS INT,
    DESTINATION_ID INT,
    DESTINATION_TYPE_ID INT,
    HOTEL_CONTINENT INT,
    HOTEL_COUNTRY INT,
    HOTEL_MARKET INT,
    IS_BOOKING INT,
    CNT BIGINT,
    HOTEL_CLUSTER INT,

    PRIMARY KEY (HOTEL_COUNTRY, USER_LOCATION_COUNTRY,
    USER_ID,DATE_TIME)
)
```
As explained in section 4.1, the primary key in Cassandra defines the way that data is distributed across different partitions and the way it is stored to disk. In the above DDL command, the column “HOTEL_COUNTRY” was set as the partition key, and the columns “USER_LOCATION_COUNTRY”, “USER_ID” and “DATE_TIME” were set as the cluster keys; the latter two were included to ensure uniqueness. Therefore, hotels that are from the same country will be stored together in the same partition, and within a partition, the records will be sorted based on the user country, user id and the date/time of the search.

After the creation of the column family, a python script was implemented to parse the data set. Each row of the data set was transformed into a separate INSERT statement and got imported into the database. Approximately 34 million records (about 4GB of data) were imported.

4.4 Query language

Database developers can interact with Cassandra using the Cassandra Query Language (CQL), a SQL-like declarative programming language designed specifically for Cassandra. Developers can use CQL either by using the Cassandra shell (CQLsh), which is by default installed with Cassandra, or by using a Cassandra driver for the chosen programming language. The information presented in this section and in sections 4.5 and 4.6 is described in [36]. Examples of the commands presented in this section can be seen in appendix B.

4.4.1 Create

In order to import data into the database, CQL provides an “insert” statement very similar to the one that the SQL provides. The basic syntax for an “insert” statement is the following:

```
INSERT INTO table_name (<<column_name_1>>,....<<column_name_n>>)
VALUES (<value_1>>,....<value_n>>) [IF NOT EXISTS]
[ USING [ TTL time_value ] | [ TIMESTAMP timestamp_value ] ]
```

The user must specify the table name, the columns into which they want to import data and the values for each specified column. In every “insert” statement, values for each column that is part of the primary key have to be supplied (but not for the other columns of the table); if not, the insert statement is rejected. Any other column of the table that is not defined in the “insert” statement takes up no space on disk. If the primary key of the new row already exists in the table, the new column values will overwrite the previous ones, except if the IF NOT EXISTS clause is specified. Additionally, the user can specify a time-to-live value in seconds; after that time expires, the row is automatically deleted from the table. Finally, Cassandra by default marks each inserted row with its time of insertion, in microseconds. However, if the user wants to associate a specific timestamp with an insertion, they can specify it in the TIMESTAMP clause.
**4.4.2 Read**

CQL supports the familiar, SQL-like “SELECT FROM WHERE” command to retrieve results. The basic structure of this command is the following:

```
SELECT * | DISTINCT partition_key | << col_name_1>>, ..., <<col_name_n>>
FROM table_name
[ WHERE col_name_1 operator value_1 [ ... AND col_name_n operator value_n ] ]
[ ORDER BY <<cluster_col_1>> [ ASC | DESC ], ..., <<cluster_col_n>> [ASC | DESC ]
[ LIMIT n ]
```

In the SELECT clause, the user can specify “*” to retrieve all the columns of a particular result set, “DISTINCT partition_key” to get a list of all the partition keys in the table, or specific column names so as to retrieve only the values of these columns. In the FROM clause, a single table’s name is specified; Cassandra does not support join operations at all, neither nested queries. Therefore, neither multiple tables nor nested SELECT statements can be specified there.

In the WHERE clause, the user can filter the results of the query by specifying a column name, an appropriate comparison operator (‘=’, ‘<’, ‘<=’, ‘>’ or ‘>=’) and a value. Multiple columns can be specified together using the AND logical operator; Cassandra does not support any other logical operators. Additionally, in the WHERE clause, the only columns that can be specified are the ones that are part of the partition key and the clustering columns (provided that the partition key is specified). If any other column is specified the query will be rejected, unless a secondary index was created on that column (secondary indexes are explained in section 4.5). In the ORDER BY clause, the clustering columns and the sorting order (ascending or descending) can be specified. Finally, in the LIMIT clause, the user can specify the maximum number of returned rows.

**4.4.3 Update**

CQL provides an “update” command, very similar to the “insert” one. The basic structure of this command is the following:

```
UPDATE table_name
[ USING TTL time_value | TIMESTAMP timestamp_value ]
SET <<column_name_1>> = value_1 [,...,<<column_name_n>> = value_n]
WHERE <<primary_column_1>> = value_1 .. AND ...<<primary_column_n>> = value_n
[ IF EXISTS ]
```

In the UPDATE clause the name of the table has to be specified. Additionally, values for the TTL and TIMESTAMP can be supplied as well, as explained in section 4.4.1. In the SET clause, a list of column names that have to be updated is specified, along with their values. In the WHERE clause, a value for each column of the primary key has to be supplied. By default, if a row does not exist in the table, the update command will insert it. This can be avoided by specifying the IF EXISTS clause.
4.4.4 Delete

CQL provides a “delete” command that allows users to delete entire rows or individual columns from a specific row. The basic structure for the “delete” command is the following:

```
DELETE [<<col_name_1>>,...,<<col_name_n>>]
FROM table_name
[ USING TIMESTAMP timestamp_value ]
WHERE <<primary_column_1>> = value_1 .. AND ...<<primary_column_n>> = | < | >
|<= | >= value_n
[ IF [NOT] EXISTS]
```

In the delete clause, the names of the columns that have to be deleted can be specified. If no column is specified, the entire row is deleted. The FROM and USING TIMESTAMPS clauses are the same as explained in section 4.4.1. In the WHERE clause however, Cassandra allows the user to specify either all the columns of the primary key or the partition key and inequality operators for the clustering columns. Finally, the optional IF EXISTS clause causes the command to fail if the specified row exists, while the IF NOT EXISTS clause causes the command to fail if the row exists.

4.4.5 Lightweight transaction support

Cassandra provides lightweight transaction support in the “insert”, “update” and “delete” commands via the IF clause. The IF clause can cause a transaction to fail if the specified criteria are not met. Lightweight transactions are useful in cases where specific checks have to be performed before modifying data (e.g., subtract an amount of money from an account balance only if the balance will not get negative after the transaction). However, they should be used sparingly as they add a considerable amount of latency to a write operation.

4.4.6 Aggregate functions

Cassandra provides the standard aggregate functions of MIN, MAX, AVG, COUNT and SUM as built-in functions. They can be used in the SELECT clause of any “read” command, and they can be applied to a single column.

4.4.7 User Defined Functions (UDF) and User Defined Aggregate functions (UDA)

As an alternative to the built-in aggregation functions, Cassandra allows the users to define their own UDFs and UDAs. After a UDF is created, it can be used in the SELECT clause of a “read” query like a standard function and it will be performed on each row of the table. The basic syntax for a UDF is the following:

```
CREATE FUNCTION [IF NOT EXISTS] function_name
( <<arg_name_1>> <<arg_type_1>>, ...,<<arg_name_n>> <<arg_type_n>> )
(CALLED | RETURNS NULL) ON NULL INPUT
RETURNS <<type>>
LANGUAGE <<language>>
```
AS <<body>>

In the CREATE FUNCTION clause the name of the UDF is stated, followed by a list of input parameters and their types. If the CALLED ON NULL INPUT clause is specified, then the function will always be executed; on the other hand, if the RETURNS NULL ON NULL INPUT clause is specified, if any of the input parameters is null, the function will always return null. In the RETURNS clause the type of the return value has to be supplied. Finally, in the AS clause the body of the function is implemented using the programming language (e.g., Java, Python) that was specified in the LANGUAGE clause.

Standard UDFs are stateless, and therefore they cannot be used for aggregate functions. On the other hand, UDAs are stateful, and they are created using standard UDFs. The basic syntax for a UDA is the following:

CREATE AGGREGATE [IF NOT EXISTS] aggregate_name
( <<arg_name_1>> <<arg_type_1>>, ..., <<arg_name_n>> <<arg_type_n>> )
SFUNC <<state_UDF_name>>
STYPE <<state_UDF_return_type>>
FINALFUNC <<final_UDF_name>>
INITCOND (<<value_1>>, ..., <<value_n>>)

In the CREATE AGGREGATE clause the name of the UDA is stated, followed by a list of input parameters and their types. In the SFUNC and STYPE clauses, the name and return type of the (stateful) UDF are specified. In the FINALFUNC clause, another UDF can be specified so as to apply a final function on the aggregated results of the previous UDF. Finally, in the INITCOND clause the initial values for the state variables can be specified.

4.5 Indexing support

As explained in section 4.4.2, Cassandra does not allow the usage of columns that are not part of the primary key in the WHERE clause of a read query. However, it allows the usage of a column on which a secondary index has been created. Therefore, in Cassandra a secondary index is a means to enhance the functionality and expressiveness of its query language, rather than a way to improve the performance. For example, creating a secondary index on the column HOTEL_CONTINENT of the table that is presented in section 4.3, will make that column eligible to be used in the WHERE clause of a query. The basic syntax for the creation of a secondary index is the following:

CREATE INDEX <<index_name>> ON <<table_name>> (<<column_name>>)

An index can be created on any column of a table, but a combination of multiple columns for a single index cannot be used.
Indexes may enhance the functionality and the query language expressiveness of Cassandra, but in some cases they can induce additional, non-negligible maintenance and query overheads. In general, indexes should not be created on high cardinality columns that have many distinct values (e.g., a column that stores the ID number of a person), neither on columns that are frequently updated or deleted. Indexes are best used on low cardinality columns, like a “country” column in a table that contains personal information about people.

4.6 Materialised views

As an alternative to secondary indexes, Cassandra provides materialised views to enhance the functionality of the query language. A materialised view is a secondary table built from another table’s data and using a different primary key so as to support a different query (than the one that the main table supports). As stated in section 4.4.2, a common practice in Cassandra is to manually maintain a separate table for each query. When a table is modified however, the client application is responsible to modify all the additional tables that hold relevant data so as to maintain data integrity between tables. Materialised views make things much easier, by updating and deleting values automatically when the primary table is modified. The basic syntax for a materialised view creation is the following:

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] <<view_name>>
AS SELECT <<column_1>>,..,<<column_n>>
FROM table_name
WHERE <<primary_column_1>> IS NOT NULL …AND… <<primary_column_n>> IS NOT NULL
PRIMARY KEY(<<primary_column_1>>,…,<<primary_column_n>>)
```

Materialised views are created by stating their name, the primary table’s name and appropriate columns as well as the new primary key. Columns that are part of the new primary key cannot be null, therefore the IS NOT NULL statement is used. Additionally, all the columns of the original table’s primary key must be part of the materialised view’s primary key, and only one extra column is allowed to be added to it. Due to the fact that materialised views reorder the way that data is stored using a different primary key, they support very efficient read queries - just like normal tables. However, write performance of the primary table may suffer when views exist, because when a row is updated in the primary table an additional operation has to be executed in order to update the corresponding row in the view.
Chapter 5

MongoDB and Cassandra quantitative evaluation

5.1 Dataset

For the quantitative evaluation of MongoDB and Cassandra, the same data set had to be imported into both databases in order to benchmark them on equal terms and to obtain objective performance results. The new data set had to be simple, so as not to favour MongoDB that supports more complex data than Cassandra. Additionally, due to the fact that database systems are primarily concerned with I/O performance, it had to be fairly large. The data set that was selected to be imported was Facebook’s user check-in data set that was provided for Kaggle’s “Facebook V: Predicting check ins” competition [38].

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
</tr>
<tr>
<td>row_id</td>
<td>int</td>
</tr>
<tr>
<td>coordinate_x</td>
<td>float</td>
</tr>
<tr>
<td>coordinate_y</td>
<td>float</td>
</tr>
<tr>
<td>time</td>
<td>int</td>
</tr>
<tr>
<td>place_id</td>
<td>long</td>
</tr>
</tbody>
</table>

Table 5.1 Data set’s attributes

This data set consists of information about the location, the time and the actual place of a check-in and it is very simple, consisting only of numeric values. A full list of the attributes of the data set can be seen in table 5.1. The data set consists of approximately 29 million records and its size is 1.2 GB. Due to the fact that it is relatively small, the data set was copied and imported eight times to each database, to produce a total of 232 million records and 9.6 GB of data. Therefore, one additional integer field (id) was
created manually to uniquely identify each record. This field served as the primary key in both databases.

5.2 Performance metrics

To assess the performance of the databases a common metric had to be used. Two metrics that are frequently used when benchmarking databases are the throughput, which is the average number of operations per second and the latency, which is the average time needed to perform a single operation. These two metrics were used in the quantitative evaluation, and the formulas used to calculate them can be seen in equations 5.1 and 5.2 respectively.

\[
\text{Throughput} = \frac{\text{(number of operations per thread)} \times \text{(number of threads)}}{\text{average execution time for 1 thread}} \quad (5.1)
\]

\[
\text{Latency} = \frac{\text{average execution time for 1 thread}}{\text{number of operations per thread}} \quad (5.2)
\]

5.3 Methodology

MongoDB and Cassandra support very different data models and query languages. MongoDB is a document-oriented database that stores data as JSON-like documents and supports a very expressive query language. On the other hand, Cassandra is a wide-column store that stores data in partitioned tables and supports a query language with very limited capabilities. While in MongoDB very complex, multistage queries can be expressed, in Cassandra they cannot. In order to benchmark both databases on equal terms, the performance of the same set of queries had to be evaluated. Therefore, the evaluation workloads consisted of simple CRUD operations that could be expressed in both databases.

5.3.1 Data import workload

This workload was used to assess the data import performance of the databases. The same dataset was imported to both databases, so as to calculate the data import throughput and latency. The query used to import the data is the following (expressed in CQL):

```
INSERT INTO data (id, row_id, coord_x, coord_y, time, place_id)
VALUES <<id_val>>, <<row_id_val>>, <<coord_x_val>>, <<coord_y_val>>, <<time_val>>, <<place_id_val>>)
```

5.3.2 Read workload

This workload was used to assess the read performance of the databases and it consisted only of read operations. The query used to calculate the throughput and the latency is the following (expressed in CQL):

```
SELECT *
```
FROM data
WHERE id = <<random_value>>

5.3.3 Write workload
This workload was used to assess the write performance of the databases and it consisted only of write operations. The query used to calculate the throughput and the latency is the following (expressed in CQL):
UPDATE data
SET row_id = -1, coord_x = -1.0, coord_y = -1.0, time = -1, place_id = -1
WHERE id = <<random_value>>

5.4 Hardware and software specifications

5.4.1 Hardware specifications
Both the qualitative and quantitative evaluations were performed on a cluster of nodes provided by EPCC. The hardware specifications for each node are presented in this section:

- **Processor**: Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
- **Number of processors**: 2 (NUMA)
- **Cores per processor**: 10
- **Cache**: 32 KB (L1), 256 KB (L2), 25.6 MB (L3)
- **Main Memory**: 132 GB
- **Storage**: 456 GB HDD (local)

5.4.2 Software specifications
In this section the specifications of all the software applications and systems that were used are presented:

- **Operating system**: CentOS Linux release 7.2.1511
- **MongoDB version**: 3.2.6
- **Cassandra version**: 3.4.0
- **Python version**: 3.4.3
- **Cassandra driver for Python version**: 3.4.1
• **MongoDB driver for Python version:** 3.2.2

### 5.5 Databases configuration

The databases were deployed on a single node, to evaluate the single server performance, and on a small cluster consisting of three nodes, to evaluate the multi-node performance and scalability. Therefore, the tests presented in section 5.2 were conducted twice, once for each deployment. In the single node evaluation, a single multithreaded script for each workload was used to measure the performance. In the multi-node evaluation however, three multithreaded scripts for each workload were executed in parallel, one for each local database instance, in order to stress the databases as much as possible.

#### 5.5.1 Cassandra configuration

For both the single and multi-node evaluation of Cassandra a keyspace (database) consisting of a single table was created. Due to the fact that only 1-3 nodes were used, the replication factor for the keyspace was set to 1; if it was not, in the cluster evaluation every node would be able to execute queries using its own data, eliminating the need for inter-node communication. Additionally, the durable writes option was set to true so as to ensure data durability and because this is the default configuration of both Cassandra and MongoDB. Instances of Cassandra were installed on 4 nodes; one was used for the single node evaluation and the other three were configured to communicate and form a cluster. In the cluster deployment, the data was automatically distributed by Cassandra using the partition key across all the nodes that participated in the cluster. For the single node evaluation, the client application had to connect to the appropriate node to execute queries. For the multi-node evaluation, the client application could connect to any node in the cluster and that node could serve as the coordinator (as explained in section 4); therefore, scripts on each node connected to the local instance of Cassandra on that node.

#### 5.5.2 MongoDB configuration

For both the single and multi-node evaluation of MongoDB a database consisting of a single collection was created. However, data distribution is not enabled by default in MongoDB; for the cluster evaluation the collection was configured manually to enable sharding - using the “_id” field as the shard key - and distribute the data across all the nodes that participated in the cluster using the default range-based sharding. Similarly to Cassandra, no secondary shards were used to store replicated data. Additionally, the default configuration for durable writes was used as well. Instances of MongoDB were installed on 4 nodes too; one was used for the single node evaluation, and the other three for the cluster evaluation. Due to the fact that client applications have to connect to a query router to execute queries, a query router was started on each of the nodes so
that each script could connect to the local one and execute the queries in parallel without any bottlenecks.

5.6 Performance results

The performance of the databases was evaluated using the set of queries presented in section 5.3. The queries were expressed in Python, using the appropriate MongoDB and Cassandra drivers. Initially, the data set was imported into both databases; consequently, the nodes were restarted in order to erase any data that resided into the main memory so as to evaluate the I/O, not the main memory performance of the databases.

The read and write workloads consisted of 100 executions of each query. In each execution, a random value was generated to be used in the WHERE clause. The total execution time for the 100 queries was measured so as to calculate the average latency and throughput. Each test was performed 10 times, on two different days; in this section, the average throughput and latency of the 20 runs are presented. The detailed results of the performance evaluation can be seen in appendix A.

5.6.1 Data import workload

In figure 5.1 the data import throughput for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. In the single node deployment, MongoDB outperforms Cassandra; it imported data twice as fast as Cassandra did for every number of threads used. In the cluster deployment, MongoDB still outperforms Cassandra, but the difference is considerably reduced.
The data import performance of the databases was expected to be more or less the same. When importing data, neither Cassandra nor MongoDB access the disk directly; instead, data is written to the main memory and consequently it is flushed to disk at a later point in time. Therefore, instead of accessing the disk every time a document (MongoDB) or row (Cassandra) is imported, the disk is accessed once to write a batch of records. Due to the fact that both databases implement this functionality, the import performance for the single node, single thread test was expected to be the same for both. However, MongoDB imported data twice as fast as Cassandra did, probably because the internal implementation of this functionality in MongoDB is more efficient than it is in Cassandra.

In the single node, 2-16 threads tests MongoDB continued to be twice as fast as Cassandra. Given the fact that MongoDB was twice as fast in the single node, single thread test and the way that concurrency control is implemented in both databases, it was expected that MongoDB would have continued to be twice as fast when multiple threads were used. Cassandra uses timestamps to implement concurrency control. When a value for a column is inserted in Cassandra, the exact time of that write operation is stored with the value. If two values for the same column are inserted concurrently, they are both stored. However, when reading data the timestamps are checked, and in cases where a column has multiple values associated with it only the most recent one is returned. Therefore, Cassandra never locks anything at all and so it provides true parallel performance. MongoDB (version 3.2 and later) uses a similar strategy to implement document-level optimistic concurrency control. Documents can be inserted concurrently in a collection and MongoDB assumes that write conflicts will not occur. Each write operation is provided with a snapshot (consistent view) of the data. If two operations attempt to write to the same document concurrently, they will be both provided with two distinct snapshots of the same data. The operation that finishes first will succeed, but the other will fail; MongoDB will transparently retry the failed operation, but it will provide it with an updated snapshot that contains the modifications that the previous operation made. Therefore, given the fact that in during the import process every document had a unique ID, it can be concluded that write conflicts did not occur and so MongoDB provided true parallel performance as well. An important thing to note is that in previous versions of MongoDB (before 3.2) concurrency control was implemented by locking at the collection level. Therefore, only one thread could write at a time and parallel writes to different documents of the same collection were not feasible.

Finally, in the cluster tests the difference in performance is considerably reduced. This is probably due to the fact that communications are implemented a bit differently in the two databases. When a new row is inserted in Cassandra, the client application can connect to any node in the cluster, and that node will serve as the coordinator and forward the request to the appropriate node that is responsible for that row. On the other hand, MongoDB does not allow applications to connect directly to the shards. Instead, applications must connect to a query router (a different process), and subsequently the query router will have to forward the data to the appropriate shard. In insert operations, both the coordinator and the query router work in a similar way; they take as input the primary key of the inserted record and they calculate the id of the partition/shard they
should forward it to. As the results indicate however, the implementation of the coordinator in Cassandra that is integrated into the main database process is more efficient than the one of the query router in MongoDB that is implemented as a separate process.

![Figure 5.2 Data import latency](image)

In figure 5.2 the data import latency for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. In the single node tests, Cassandra has twice as much latency as MongoDB but in the cluster tests the difference is considerably reduced. This was expected for the reasons explained for figure 5.1. In general, the latency is stable for both databases when the number of threads is eight or less. When 16 threads are used, the latency is increased. From this observation it can be concluded that eight threads are the threshold where throughput is increased while latency remains low for this particular hardware and software configuration. When more than eight threads are used, more throughput can be achieved at the expense of higher latency for a single operation.
5.6.2 Read workload

In figure 5.3 the read workload throughput for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. Both databases have more or less the same performance, with MongoDB having slightly more throughput in the single node deployment and Cassandra having slightly more throughput in the cluster deployment.

Both databases have significantly lower read than data import performance and that was expected. During data import, the databases need only to access the main memory; the data is then asynchronously written to disk at a later point in time. In every read request however, both Cassandra and MongoDB have to access the disk, unless the requested data was transferred to the main memory earlier during the execution of another query.

The two databases use different approaches to provide efficient read queries. MongoDB can index a field that is used in a read query (the primary key field “_id” is indexed by default) using B-tree data structures. The values of an indexed field are stored in the leaves of the tree along with a pointer that points to the starting position of the record on disk. The algorithmic complexity of a B-tree search is $O(\log_mN)$; in database systems, the base of the logarithm is relatively large, about 100. Therefore, even if one billion records are stored, using the B-tree index the position of the requested record can be found in 4-5 steps. Cassandra takes a different approach; instead of B-trees, it uses bloom filters. Data in Cassandra are stored in the form of Sorted String Tables (SSTables). Within an SSTable, data is sorted based on the primary key and each SSTable has its own bloom filter. The bloom filter uses a set of hash functions (used on the primary key) to indicate whether a given record is part of an SSTable or not. The algorithmic complexity of a bloom filter search is $O(k)$, where $k$ is the number of hash
functions. Multiple hash functions can be applied together in parallel, so bloom filters can indicate whether a record is stored in a specific SSTable or not in constant time.

By observing figure 5.3, it can be concluded that MongoDB’s B-tree mechanism is slightly more efficient than Cassandra’s bloom filters because the single node performance of MongoDB is slightly better. Cassandra, however, performs slightly better in the cluster deployment, and this is possibly due to a more efficient communications implementation, as explained in section 5.6.1.

In figure 5.4 the read latency for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. Cassandra has higher latency than MongoDB in the single node deployment but lower latency in the cluster deployment due to the reasons explained for figure 5.3.

In this graph however, it can be observed that the latency for both databases is increased along with the number of threads, as it was expected. When reading data, most of the times both databases need to access the disk. The disk cannot be accessed by multiple threads concurrently; only one thread at a time can access it. Therefore, while a thread is doing disk I/O to retrieve the results for a query, all the others must wait, and this adds a considerable amount of overhead in the execution time of a single operation.

Finally, in this graph the latency observed in the cluster evaluation is lower than the one observed in the single node evaluation. Given the fact that more queries are executed in total in the cluster evaluation (48 threads in total instead of 16 executing queries), there
is a higher chance of some queries accessing data in main memory (instead of disk) that were fetched by earlier ones, and possibly for this reason the latency is lower.

5.6.3 Write workload

![Figure 5.5 Write throughput](image)

In figure 5.5 the write workload throughput for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. Cassandra outperforms MongoDB by orders of magnitude on both the single node and cluster deployments, and it was expected because the two databases implement update operations very differently.

MongoDB updates data in place; that is, updates modify the original document, not a copy of it. If a document has to be updated and is not in main memory, it has to be fetched from disk. By observing figures 5.1 (no disk access) and 5.3 (a lot of disk accesses), it can be concluded that every time an operation needs to access the disk, the performance drops dramatically; MongoDB achieved more than 70000 ops/sec in figure 5.1, but less than 500 ops/sec in figure 5.3. On the other hand, Cassandra never actually updates the real data. As explained in section 5.6.1, Cassandra assigns a timestamp to every write operation. When an update occurs, the old values are never modified but instead new ones are inserted with a newer timestamp. When data is read, only the values with the latest timestamps are returned. Cassandra actually deletes outdated data during a procedure called compaction, which is performed at regular time intervals. Therefore, Cassandra never needs to access the disk when updating data; all updates are written to the main memory. This implementation allows Cassandra to achieve the same performance for updates as it achieves for writes, which is orders of magnitude faster than MongoDB’s update performance.
In figure 5.6 the write workload latency for MongoDB and Cassandra for single and three nodes deployments is presented, using 1-16 threads per node. As it was expected, MongoDB’s latency is orders of magnitude higher than Cassandra’s, due to the reasons explained for figure 5.5. An interesting observation in this figure is that the latency for the cluster deployment of MongoDB is lower than the one for the single node. Given the facts that updates in MongoDB need to fetch the data from disk (if not present in the main memory) and that more queries are executed in total in the cluster evaluation, there is a higher chance of some queries accessing data in main memory that were fetched by earlier ones, and possibly for this reason the latency is lower, as explained in section 5.6.2.
Chapter 6

Amazon DynamoDB qualitative evaluation

Amazon DynamoDB is a proprietary, fully managed NoSQL Database as a Service (DBaaS) that is offered and hosted by Amazon within the AWS infrastructure. DynamoDB is essentially a key-value store and it was released in 2012. It is currently the most popular DBaaS, the third most popular key-value store and the 25th most popular database management system.

6.1 Data model

DynamoDB organises data in tables, items and attributes, as illustrated in figure 6.1. Tables in DynamoDB are collections of items, as tables in relational databases are collections of rows. Each item can consist of an arbitrary number of attributes, but the maximum size of an item cannot exceed 400 KB. Each attribute consists of a name (key) and a value. The supported types for the attribute values are number, string, boolean, binary, set, list and map.

Figure 6.1 DynamoDB data model
Tables can have infinite size, and each one must have a primary key to uniquely identify each item. The primary key can be simple, consisting of a single attribute (hash key) or composite, consisting of two attributes (hash and range keys). The hash key is used to determine to which partition each item will be stored, and the range key is used to sort the items within a partition. All the attributes that form the primary key must exist in every item of a table. Other than the definition of the primary key, a schema does not need to be defined up-front for a table, and different items in the same table do not need to have the same number of attributes. Therefore, DynamoDB provides high schema flexibility [39].

6.2 Architecture

6.2.1 Partitions and data distribution

DynamoDB stores data in partitions, allocated storage on solid-state drives (SSDs). DynamoDB automatically allocates sufficient partitions for a table and replicates each partition across multiple availability zones within an AWS region during the creation of the table. The number of partitions that are initially allocated depends on the provisioned throughput requested by the user for that table (more about the provisioned throughput in section 6.6). Each partition can store up to 10GB of data. If the storage size limit is reached or if the user increases the provisioned throughput, DynamoDB will automatically allocate more partitions and re-distribute the data.

Figure 6.2 Data distribution using the hash and range keys [40]
When an item is written to the table, DynamoDB hashes the value of the hash key using an internal hash function. The output of the hash function determines the partition to which the item will be stored. If a composite primary key is used, within a partition all the items with the same hash key will be stored physically close together and they will be sorted based on the range key [40]. In figure 6.2 an example of the data distribution in DynamoDB can be seen. Items with the same hash key (“AnimalType”) are stored on the same partition, and within the partition are sorted based on the range key (“Name”).

6.2.2 Availability, Consistency and Durability

DynamoDB provides high availability by storing three geographically distributed replicas of each table. Data durability is ensured by synchronously replicating writes across the three replicas. If a replica fails at some point, data can be written and read from the other two. DynamoDB by default supports eventual consistent reads, to maximise throughput. However, it allows the users to specify the consistency level for each read operation and therefore it can provide strong consistency as well [39].

6.3 Data import

In order to evaluate the features of DynamoDB, a data set had to be imported. Amazon DynamoDB is included in the AWS free tier of services, but the provided standard throughput is very low (25 write/read operations/sec). Therefore, the data set had to be small in order to avoid spending too much time in the import process. The data set that was selected to be imported was the Amazon fine foods reviews [41]. This data set was collected for academic research purposes, and it is publicly available to use by citing the paper of the people that collected the data [42]. It has a relatively small size (300 MB), and it consists of about 500,000 user reviews of fine foods from Amazon. A full list of the data set’s attributes can be seen in table 6.1

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Int</td>
</tr>
<tr>
<td>ProductId</td>
<td>String</td>
</tr>
<tr>
<td>UserId</td>
<td>String</td>
</tr>
<tr>
<td>ProfileName</td>
<td>String</td>
</tr>
<tr>
<td>HelpfulnessNumerator</td>
<td>Int</td>
</tr>
<tr>
<td>HelpfulnessDenominator</td>
<td>Int</td>
</tr>
<tr>
<td>Score</td>
<td>Int</td>
</tr>
<tr>
<td>Time</td>
<td>String</td>
</tr>
</tbody>
</table>
Before importing data in DynamoDB, a table has to be created. During the creation of a new table, the attributes that will serve as the hash and range (if a range key exists) keys must be specified, as well as the provisioned throughput for the table. The provisioned throughput is used to specify how many operations/second the table will be capable of executing. For the table that was created to import the reviews, the “id” attribute served as the hash key; a range key was not used. The provisioned throughput was set to 25 read and write operations/second, the maximum available that is free. After the creation of the table, a python script was implemented to parse and import the data into the database.

### 6.4 Query language

Database developers can interact with DynamoDB by using the appropriate DynamoDB driver in a high-level programming language (e.g., Java, Python, JavaScript). In this section, the basic CRUD operations as well as some more advanced querying operations that are supported by DynamoDB are presented. The information presented in this section is described in [43]. Examples of the commands presented in this section can be seen in appendix B.

#### 6.4.1 Create

DynamoDB provides the “put_item” command to insert data into a table. The basic structure of the command is the following:

```python
put_item(
    TableName = “tablename”
    Item = {
        ‘attribute_1_name’: attribute_1_value,
        ‘attribute_n_name’: attribute_n_value
    }
)
```

In the TableName clause the name of the table into which the data will be imported must be specified, and in the Item clause the attributes of the item that will be imported have to be specified as well. Items in the same table do not need to have the same attributes; however, every item must specify the primary key of the table. If not, the operation will fail.

#### 6.4.2 Read

DynamoDB provides the “get_item” command to read data from a table. The basic structure of the command is the following:
get_item(
    TableName = ‘tablename’,
    Key = {
        ‘hash_key_name’: hash_key_value,
        ‘range_key_name’: range_key_value
    }
)

In the TableName clause the name of the table’s from which the data will be retrieved must be specified, and in the Key clause the values of the attributes that form the primary key must be specified as well. If a simple primary key is used only the hash key must be specified; if a composite primary key is used, both the hash and range keys must be specified. No other attributes can be specified in the “get” command; data can be retrieved only by the primary key.

6.4.3 Update

DynamoDB provides the “update_item” command to update an item in a table. The basic structure of the command is the following:
update_item(
    TableName = ‘tablename’,
    Key = {
        ‘hash_key_name’: hash_key_value,
        ‘range_key_name’: range_key_value
    },
    UpdateExpression = ‘update expression’,
    ExpressionAttributeValues = {expression attribute values}
)

Values for the TableName and Key clauses must be supplied as explained in section 6.4.3. In the UpdateExpression clause, the user can specify the attributes they wish to add, modify or delete from an Item, and in the ExpressionAttributeValues the user can specify new values for those attributes.

6.4.4 Delete

DynamoDB provides the “delete_item” command to delete an item from a table. The basic structure of this command is the following:
delete_item(
    TableName = ‘tablename’,
    Key = {
        ‘hash_key_name’: hash_key_value,
        ‘range_key_name’: range_key_value
    }
)
Values for the TableName and Key clauses must be supplied as explained in section 6.4.3. Once an item is deleted, it is removed permanently and it cannot be restored. One example of a “delete” command is the following:

```
define_item(
    TableName = 'reviews',
    Key = {
        'id': -1
    }
)
```
This command deletes the item with “id” equal to -1 from the table “reviews”.

**6.4.5 Scan**

DynamoDB supports a “scan” command, which reads and returns every item of a table. This operation supports filtering to narrow down the number of the returned results, but the filtering expression is applied after the entire table has been scanned. The basic structure for this command is the following:

```
scan(
    TableName = 'tablename',
    FilterExpression = 'filter expression',
    ExpressionAttributeValues = {expression attribute values},
    ProjectionExpression = 'projection expression'
)
```
A value for the TableName clause must be supplied as explained in section 6.4.3. In the FilterExpression clause, the expression that will be applied to filter the results is specified along with the attributes that will be applied on, and in the ExpressionAttributeValues, the values for those attributes are specified. Finally, in the ProjectionExpression clause, a list of the attributes that will be returned can be specified.

**6.4.6 Query**

DynamoDB allows the users to retrieve all the items or a subset of the items of specific partitions, using the “query” command. This command works similarly to the “scan” one, but instead of scanning the entire table, it can use the hash and range keys of a table or an index (more about indexes in section 6.5) to scan only specific partitions. The basic structure for this command is the following:

```
query(
    TableName = 'table name',
    IndexName = 'index name',
    FilterExpression = 'filter expression',
    KeyConditionExpression = 'key condition expression',
    ExpressionAttributeValues = {expression attribute values},
    ProjectionExpression = 'projection expression'
)
A value for the TableName clause must be supplied as explained in section 6.4.3. Additionally, an index name can be specified in the IndexName clause, to query an index of the specified table instead of the table itself. In the FilterExpression and ProjectionExpression clauses values can be supplied as explained in section 6.4.5. In the KeyConditionExpression, the expression that will be used on the hash and range keys in order to scan only specific partitions must be supplied. Finally, in the ExpressionAttributeValues clause the values for all the attributes that are used in the KeyConditionExpression and FilterExpression clauses must be supplied.

6.5 Indexing support

As explained in section 6.4.6, when using the “query” command to retrieve a set of items, only the keys of the table can be supplied in the query expression; any other attribute of the items cannot be used. To overcome this limitation and enhance the functionality of its query language, DynamoDB supports two kinds of secondary indexes, Local Secondary Indexes (LSI) and Global Secondary Indexes (GSI).

LSIs are data structures that contain copies of the items of a table and they physically reorder them, using the same hash key but different range key. Not every attribute of the items of the main table has to be copied to a LSI; the user can specify which attributes will be projected. When an operation modifies an item in the main table, the corresponding item in the LSI is automatically modified as well. LSIs can be created during the creation of a new table, and after their creation data can be queried by using the LSI’s range key instead of the main table’s one.

GSIs are very similar to LSIs; they contain copies of the items of a table, they user can specify which attributes will be projected into a GSI and when items in the main table are modified, the corresponding ones in the GSI are automatically modified as well. However, a GSI will reorder the items of the main table using different hash and range keys. GSIs can be created during or after the creation of a new table, and once created, data can be queried using the GSI’s hash and range keys instead of the main table’s ones. Currently, up to five GSI’s and LSI’s can be created for a table.

6.6 Cloud platform characteristics evaluation

6.6.1 Ease of use

DynamoDB is a fully managed DBaaS and is hosted within the AWS infrastructure. Database administrators do not need to set up, configure and maintain any hardware or software resources. Instead, they only specify the rate of read and write throughput they wish for each table to achieve. DynamoDB will then automatically handle the management of resources to provide the requested throughput rate. Storage capacity does not need to be provisioned; DynamoDB will automatically allocate additional storage resources as the volume of data increases. In general, configuring DynamoDB is very easy; other than the required throughput, nothing else has to be configured manually.
6.6.2 Throughput and storage costs

Database owners pay a flat, hourly rate based on the throughput capacity they provision for each table. They can purchase throughput in terms of read and write units. A write unit equals to a write operation per second for items up to 1KB, and a read unit equals to one strongly consistent or two eventually consistent reads per second for items up to 4KB; larger items will consume more read/write units. Regarding storage requirements, DynamoDB provides 25GB for free; if that limit is exceeded, a flat monthly rate for each extra GB of storage is billed. The throughput and storage prices vary slightly across different AWS regions; the exact prices for the region in which the data for this evaluation was hosted (EU-Ireland) can be seen below:

- **Read:** $0.000147 per unit per hour
- **Write:** $0.000735 per unit per hour
- **Storage:** $0.283 per GB per month

Database owners can increase or decrease the rate of the provisioned throughput for any table any time. However, if they want to save money they can purchase reserved capacity throughput. That is, they commit to pay for a minimum amount of throughput capacity for one or three years, and they can save up to 53% and 76% correspondingly [44].

In section 5, MongoDB’s and Cassandra’s performance was evaluated. In table 6.2 the monthly prices that a DynamoDB database owner would have to pay in order to achieve the minimum and maximum throughput rates that were measured for MongoDB and Cassandra (without purchasing reserved throughput capacity) can be seen.

<table>
<thead>
<tr>
<th>Database</th>
<th>Read throughput (ops/sec)</th>
<th>Write (update) throughput (ops/sec)</th>
<th>DynamoDB monthly price</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node, 1 thread)</td>
<td>61</td>
<td>38</td>
<td>$9.62</td>
</tr>
<tr>
<td>MongoDB (3 nodes, 48 threads)</td>
<td>494</td>
<td>333</td>
<td>$191.28</td>
</tr>
<tr>
<td>Cassandra (1 node, 1 thread)</td>
<td>62</td>
<td>2040</td>
<td>$1529.73</td>
</tr>
<tr>
<td>Cassandra (3 nodes, 48 threads)</td>
<td>504</td>
<td>67663</td>
<td>$34759.31</td>
</tr>
</tbody>
</table>

**Table 6.2** DynamoDB prices to achieve the minimum and maximum throughput achieved by MongoDB and Cassandra as presented in section 5

6.6.3 Elasticity and on-demand self-service

Amazon allows DynamoDB database owners to manage and monitor the usage of their databases through the AWS console. The console provides useful information such as
provisioned and consumed read and write throughput on a per table and index basis, and get, put, query and scan average, minimum or maximum latency. Database owners can monitor the usage of each table or index individually and can increase or decrease the provisioned throughput accordingly. The provisioned throughput for a table or index can be set and updated at any time either via the AWS console or via the DynamoDB API in a programming language. The provisioned throughput can be modified without human intervention at all and can be increased up to 10000 operations per second. For higher throughput rates, AWS users have to contact Amazon for approval.

6.6.4 Response time

As stated in section 6.6.3, the AWS console provides information about the latency of basic CRUD operations. After executing a small set of random puts and gets, the average latency rate per operation that was provided by the AWS console was between 30-40 milliseconds. However, the latency that was measured by the client application that was used to perform the puts and gets was more than an order of magnitude higher; each operation took approximately half a second to complete. This difference in the latency can be attributed to the fact that while the AWS console displays the actual latency, from the time the database receives a request until the time it provides a response, the client application measured the latency from the time the request was issued, until the response was received from the database. Therefore, the latency measured by the client application includes the time required for the request to be sent to the AWS region that the DynamoDB instance is hosted and the time that the response takes to be received. This additional network latency causes the actual response time to be much higher than it would have been if a database was deployed locally instead of on the cloud.
Chapter 7

MongoDB, Cassandra and Amazon DynamoDB comparison

In chapters 3, 4 and 6 MongoDB, Cassandra and Amazon DynamoDB are qualitatively evaluated, and in chapter 5 MongoDB and Cassandra are quantitatively evaluated. In this chapter, the three databases will be compared qualitatively using the following quality attributes: ease of use, availability, scalability, data durability, consistency, data model and schema flexibility, query language expressiveness and secondary indexing support. In addition to that, MongoDB will be quantitatively compared using the following performance metrics: data import performance, read performance and write (update) performance. In table 7.1 the qualitative and quantitative characteristics of the databases are summarised.

<table>
<thead>
<tr>
<th></th>
<th>MongoDB</th>
<th>Cassandra</th>
<th>DynamoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td>Document</td>
<td>Wide column</td>
<td>Key-value</td>
</tr>
<tr>
<td>Availability</td>
<td>High</td>
<td>Very high</td>
<td>Very high</td>
</tr>
<tr>
<td>Scalability</td>
<td>Horizontal</td>
<td>Horizontal</td>
<td>Horizontal</td>
</tr>
<tr>
<td>Data durability</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Consistency</td>
<td>Configurable (default strong)</td>
<td>Configurable (default eventual)</td>
<td>Configurable (default eventual)</td>
</tr>
<tr>
<td>Schema flexibility</td>
<td>Very flexible</td>
<td>Rather flexible</td>
<td>Flexible</td>
</tr>
<tr>
<td>Query language expressiveness</td>
<td>Very high</td>
<td>Low</td>
<td>Very low</td>
</tr>
<tr>
<td>Secondary indexing support</td>
<td>Yes</td>
<td>Limited</td>
<td>Limited</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Rather easy</td>
<td>Easy</td>
<td>Very easy</td>
</tr>
</tbody>
</table>

51
<table>
<thead>
<tr>
<th></th>
<th>Left outer join supported</th>
<th>No</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactions</td>
<td>No</td>
<td>No (lightweight transactions supported)</td>
<td>No</td>
</tr>
<tr>
<td>Materialised views</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DBaaS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Aggregate functions</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>User Defined Functions</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Data import performance</td>
<td>High</td>
<td>High</td>
<td>-</td>
</tr>
<tr>
<td>Read performance</td>
<td>Low</td>
<td>Low</td>
<td>-</td>
</tr>
<tr>
<td>Write (update) performance</td>
<td>Low</td>
<td>High</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7.1 Summary table with the characteristics of the evaluated databases

7.1 Quality attributes

7.1.1 Availability

All three databases provide high availability by replicating data across multiple machines, though using different strategies. MongoDB maintains a primary replica of the data which by default serves all reads and writes, and multiple secondary ones that serve as copies of the primary. When the primary fails, a secondary replica must be elected to act as the new primary. On the other hand, Cassandra does not use primary and secondary replicas; instead, any node in a replica set can serve any read or write request; data are replicated across the other nodes of the set in parallel. Therefore, if a node fails at some point the database will not suffer from any downtime as data can be read from and written to other nodes in the replication set. Similarly to Cassandra, DynamoDB synchronously replicates data to three different facilities in an AWS region. If a node fails, the database can still operate normally by using replicated data from the other two facilities. To sum up, it can be concluded that although all three systems provide high availability, Cassandra and DynamoDB can provide true continuous uptime as they have a master-less architecture, while MongoDB may face some brief downtime between the time that a failure occurs and the time a secondary replica assumes the role of the primary.
7.1.2 Scalability

All three databases provide high horizontal scalability and they can scale across multiple nodes. MongoDB organises data in shards, and each shard can be stored on a different machine. When data has to be read and written, a query router determines where the data has to be read from or written to and forwards the request to the appropriate shard. Cassandra and DynamoDB organise data in partitions and each partition can be stored on a different machine as well. When data has to be read or written, an internal hash function is used to calculate a value which determines to which partition the request must be forwarded to.

7.1.3 Data durability

Single server instances of MongoDB and Cassandra ensure data durability by using data logging mechanisms (the journal in MongoDB and the commit log in Cassandra). These mechanisms keep an ordered set of operations that took place in the database and they are periodically flushed to disk. In cases where hardware failures occur, when the server is up again the log files can be checked to recover and re-apply again any lost operations. In cluster deployments of MongoDB and Cassandra durability is further ensured by replicating data across multiple nodes. This is also the default behaviour of DynamoDB, as data is always synchronously replicated across three different facilities within an AWS region. To conclude, all three databases support data durability.

7.1.4 Consistency

All three database systems support configurable consistency; it can be either strong or eventual. In MongoDB, strong consistency is the default; it can be configured to eventual in cases where slightly outdated data are acceptable in order to improve the performance. On the other hand, in both Cassandra and DynamoDB eventual consistency is the default; it can be configured to strong if slightly outdated data cannot be accepted at the expense of additional latency. Therefore, it can be concluded that MongoDB has the edge over Cassandra and DynamoDB; it does not need to be configured to provide strong consistency, it is the default.

7.1.5 Schema flexibility

The highest schema flexibility is provided by MongoDB; a schema does not have to be defined before importing data into a collection, and documents in the same collection do not need to have the same attributes. DynamoDB provides flexible schemas as well, but not as flexible as MongoDB. Before importing data into a DynamoDB table, its primary key must be defined. Additionally, each item of the table must provide values for all the attributes that form the primary key. However, other attributes do not have to be defined up-front and therefore, different items in the same table may not have any common attributes other than the primary key. Finally, Cassandra provides the lowest schema flexibility of the three; a schema has to be well defined before importing data into a table by specifying the primary key and the column names and types using DDL commands. However, new columns can be added on-the-fly to a table (again, using DDL commands) without any overheads and without affecting previously inserted
rows. Additionally, each row must provide a value only for each column that is part of the primary key; any other column that was defined in the schema may or may not exist in a row. Therefore, Cassandra provides some schema flexibility as well.

7.1.6 Query language expressiveness

MongoDB supports by far the most expressive query language; it is almost as expressive as traditional SQL. All the attributes of a document can be used to query a collection and multiple attributes can be used together in the same query using various operators (e.g. comparison, logical operators). Additionally, MongoDB provides the MapReduce and the aggregation pipeline frameworks, which can perform complex, multistage queries that transform the documents into aggregate results. In a few words, almost every operation than can be expressed in SQL can be also expressed in MongoDB’s query language; MongoDB even supports left outer joins, but it does not support any other join operation neither transactions. On the other hand, Cassandra’s and DynamoDB’s query languages are not so expressive. Data in Cassandra can be queried using only the columns that form the primary key. The logical operator “AND” and the standard comparison operators are supported and can be used, as well as standard aggregate and user defined functions. DynamoDB has the least expressive query language; similarly to Cassandra, data can be queried using only the columns that form the primary key. However, aggregate and user defined functions are not supported at all.

7.1.7 Secondary indexing support

Secondary indexes in MongoDB are very similar to the ones used in relational databases; they are B-tree data structures that are used to improve query performance, and they can be created on any attribute or combination of attributes of a document of a collection. MongoDB supports a great variety of different indexing options that offer different functionalities (e.g., unique indexes to prevent duplicate values). On the other hand, secondary indexes in both Cassandra and DynamoDB are a means to enhance the functionality of the query language; when an index is created on an attribute in either Cassandra or DynamoDB, that attribute can be used to query the data. Therefore, it can be concluded that MongoDB has the highest support for secondary indexes.

7.1.8 Ease of use

DynamoDB is by far the easiest to use. The database administrator does not need to configure or think about any hardware resources at all; they only need to specify the performance they want to achieve and the system will take care of the rest. Its query language is very simple to use and easy to learn as well as it mostly consists of basic get and put operations. Cassandra and MongoDB may need to be set up and configured manually, but the procedure is very easy. To set up a Cassandra cluster, an instance of Cassandra must be installed on every node that will participate in the cluster; consequently, some parameters must be set in Cassandra’s configuration file on each node in order to allow the instances discover and connect with one another. Programming Cassandra is easy as well; CQL is a subset of SQL and therefore, users who are already familiar with SQL can use CQL with ease. Setting up a MongoDB cluster is slightly more complicated. Instances of MongoDB have to be installed on
each node that will participate in the cluster, but additionally, a configuration server and
one or more query routers have to be installed as well. The cluster can be configured
via the configuration server, by passing the IP addresses of all the nodes to which
MongoDB instances are installed. Programming MongoDB is slightly more difficult
than Cassandra and DynamoDB as well. MongoDB supports a very expressive query
language, almost as expressive as SQL, but the commands are expressed very
differently. Therefore, users who are familiar with SQL may need some time to learn
and adapt to MongoDB’s query language.

7.2 Performance metrics

In section 5 MongoDB and Cassandra were evaluated quantitatively using the data
import, read and write (update) performance as the evaluation metrics. Two evaluations
were performed for each database; one for a single node deployment, and another for a
cluster deployment.

7.2.1 Data import performance

Both databases had high data import performance; they both achieved throughput rates
up to about 70000 operations per second (MongoDB achieved about 73000 ops/sec,
while Cassandra achieved about 67000 ops/sec). MongoDB achieved twice as much
throughput as Cassandra in the single node deployment, but the difference was
considerably reduced in the cluster deployment, mainly due to the fact that inter-node
communications are implemented more efficiently in Cassandra. This high
performance was achieved because neither database accesses directly the disk in every
data import operation; instead, data is written initially to main memory and at a later
point in time it is written to disk sequentially. Therefore, the disk is accessed only once
for a large batch of insert operations.

7.2.2 Read performance

The read performance of both databases was considerably lower than the data import
performance; Cassandra managed to achieve only 504 operations per second, while
MongoDB achieved merely 494 operations per second in the best case. This was due to
the fact that for every read operation, the databases need to access the disk. MongoDB
slightly outperformed Cassandra in the single node deployment but Cassandra slightly
outperformed MongoDB in the cluster deployment.

An important thing to note however, is that in the cluster deployment the data in both
Cassandra and MongoDB was distributed in an optimal way so as to support efficient
execution of the benchmark’s queries. Cassandra by default enforces the optimal
distribution of a table’s data as data can be queried only by the columns that form the
partition key, as explained in section 4.1: each query is guaranteed to access only a
single partition. On the other hand, MongoDB does not enforce such thing; if the data
are queried using any attribute other than the shard key, every shard of the cluster will
have to be queried and the responses from all the shards will have to be gathered
together and sent back to the client application. In conclusion, the read performance
that was achieved by MongoDB would have been much worse if its advanced query frameworks, operators or attributes other than the shard key were used.

7.2.3 Write performance

Finally, Cassandra’s write performance was orders of magnitude higher than MongoDB’s in both the single node and cluster deployments. This was due to the fact that updates in MongoDB are in place; data must be fetched from the disk before it can be updated and therefore, every write operation requires a disk access. In Cassandra however, writes are not in place; the original data are never modified but instead get outdated using timestamps. Therefore, data can be written directly to the main memory eliminating the need for a disk access for every write operation.
Chapter 8

Conclusions and future work

8.1 Conclusions

In this dissertation project MongoDB, Cassandra and Amazon DynamoDB were evaluated qualitatively; the former two were evaluated quantitatively as well. Most NoSQL databases evaluations follow either a quantitative or a qualitative approach; in this project both a quantitative and qualitative approach were followed so as to obtain a clearer and complete understanding of the evaluated systems. The ultimate goal was to investigate the main features, strengths and weaknesses and to identify and recommend suitable use cases for each database.

For the qualitative evaluation of MongoDB and Cassandra, instances of both databases were installed on a single node. Then, using the Python programming language, appropriate scripts were implemented to import data and investigate the main features of them. The qualitative evaluation was followed by a quantitative one. The instances of both databases were installed on three additional nodes, so as to evaluate both the single node and cluster performance of the databases. After the installation and configuration of both databases on each node, multithreaded Python scripts were implemented to benchmark and assess the data import, read and write performance of the databases on both the single node and cluster deployments. The performance results indicated that in single server and small cluster deployments, the difference in data import and read performance of the two databases is small (although MongoDB achieved twice as much throughput as Cassandra did for the data import workload in the single node deployment, in the cluster deployment the difference was considerably reduced). However, the write performance of Cassandra is orders of magnitude higher than the one of MongoDB for both the single node and small cluster deployments. Therefore, having similar read and data import performance, both databases can be used for insert-heavy and read-heavy workloads; however, for write-heavy workloads Cassandra should be preferred.

Amazon DynamoDB was only qualitatively evaluated; being a DBaaS that is hosted within the AWS infrastructure instead of a local cluster, a quantitative comparison between DynamoDB and the other two databases would not have been objective. In order to interact with DynamoDB and evaluate its features, an AWS account was
created. DynamoDB is included in the free tier of the AWS (using limited resources however) and therefore, Python scripts were implemented again to connect to the appropriate AWS region, import data into DynamoDB and investigate its main features. In this section the main features, strengths and weaknesses of the three databases are summarised and suitable use cases for each database are recommended.

8.1.1 MongoDB characteristics summary and suitable use cases

MongoDB is a document store that provides high availability, horizontal scalability, data durability and by default strong consistency, although the latter can be configured to be eventual to further improve the performance. High availability is important in scenarios where the system must be able to serve the users continuously, without facing any downtime; horizontal scalability is required when the volume of the data grows so large that using a single machine to store and process them is not efficient anymore; data durability is very important in scenarios where data must not be lost, even when power failures occur; strong consistency is required when every user need to see the latest version of the data every time they access the database. MongoDB is completely schema-free and its query language is almost as expressive as traditional SQL. Even though it supports left outer joins, it does not support any other join operations (e.g., inner join, full join), neither transactions. It also provides a great variety of secondary indexing options. It is fairly easy to set up (for example, a cluster consisting of a few nodes can be set up in hours), maintain and understand. It provides very high data import performance but considerably lower read and write (update) performance.

MongoDB is a very good choice for use cases where the database schema cannot be known beforehand or it has to be flexible to support documents that do not share many common attributes. Additionally, it is suitable for use cases where the high expressiveness and powerful functionality of traditional SQL is required, as most SQL command can be expressed in MongoDB as well. Being ACID-compliant to the document level, MongoDB can be a good alternative to a relational database when the volume of the data grows so large that horizontal scalability becomes the most efficient and cost effective solution.

An example of a use case where MongoDB could be a good choice is for an online retail shop’s recommender system. A profile for each customer could be modelled as a document, and the history of their purchased items could be modelled as an array of embedded documents. Any data processing required to implement the recommender system could be done using the very expressive aggregation pipeline or the MapReduce framework.

Another example of a use case where MongoDB could be a good choice is for a text analysis system that analyses data from multiple different sources (e.g., Twitter tweets, Facebook comments etc). Being totally schema-free, MongoDB can store in the same collection data that come from different sources and using the text search and aggregation pipeline frameworks the users can perform the required analytic operations.
Being by default strongly consistent, MongoDB can be a good choice for use cases where users must never see outdated data. Therefore, it can be used in a massive real time game (e.g., clash of clans) where specific objectives can be available or unavailable to some players depending on the activity of other players.

MongoDB is a general purpose NoSQL database that can be used in most use cases, but it is not the best solution for every type of problem. MongoDB cannot be used in use cases where ACID-based database transactions are required (e.g., financial transactions that must move money between bank accounts); in such cases, a relational database would be a better solution. Additionally, MongoDB would not be the best solution in use cases where the queries are simple and the query access pattern is known in advance. For such use cases, Cassandra would be the ideal choice because it stores data to disk in an optimal manner as to support efficient and fast executions of specific queries. Finally, due to the fact that MongoDB may face some brief downtime between the time that a node failure occurs and the time that a secondary replica assumes the role of the primary (that is stored on the faulty node), in use cases where true continuous availability is required, a master-less database such as Cassandra or DynamoDB should be used instead.

8.1.2 Cassandra characteristics summary and suitable use cases

Cassandra is a wide column store that provides very high, continuous availability, horizontal scalability, data durability and by default eventual consistency, although the latter can be configured to be strong (at the expense of additional latency). It is very easy to set up (for example, a cluster consisting of a few nodes can be set up in minutes), maintain and understand, and it can provide very high data import and write (update) performance but considerably lower read performance. Cassandra is not totally schema-free, as tables should be created and their columns should be well defined before data could be imported. However, different rows of a table may have a different number of columns; if a value is not supplied for a column, that column will not appear at all in the specific row and it will not take any space on disk. Additionally, new columns can be added to a row on-the-fly without any cost. Cassandra’s query language is a subset of SQL but its expressiveness is considerably lower, and its secondary indexing support is limited. Cassandra imposes these query language expressiveness limitations in order to support and favour high availability, scalability and performance over complex ad-hoc querying.

Cassandra is a very good choice in use cases where true continuous availability is indispensable and where the approximate schema of the data and the query access patterns are known in advance. Cassandra does not support ad-hoc queries and its query language is very limited, so it should not be used in use cases that require complex analytical data processing. If analytical data processing is required however, Cassandra can be integrated, for example, with Hadoop or Spark and any complex operations can be performed using those frameworks. Designed to be a fully distributed data store, it is the best choice for very large volumes of data. Additionally, having an impressive write performance it could be the ideal choice for applications that perform lots of writes.
An example of a use case where Cassandra could be a good choice is for managing the user reviews of an online site that provides reviews for various restaurants, bars, coffee shops etc. By setting the place id as the partition key, Cassandra would store all the reviews for a particular place together, in the same node. Consequently, reviews within a partition could be sorted based on the date/time or number of up-votes. When a user would like to see the reviews for a particular place, Cassandra can retrieve all the data by querying only one partition and by accessing the disk only once.

Another example of a use case where Cassandra could be a good choice is for the message platform of an online application with millions of users. By setting the user id as the partition key, Cassandra would store all the messages received by a particular user together on the same partition and therefore, inbox search and message retrieval operations could be done very efficiently and quickly.

Having an impressive write (both import and update) performance, Cassandra could be a very good choice for applications that perform write-heavy workloads. For example, Cassandra could be a suitable solution for location-based applications that update the location of the user frequently. Due to the fact that Cassandra does not access the disk in every write operation, location updates can be performed very fast.

Cassandra may be the ideal database to use in many use cases, but it is not a suitable solution for every type of problem. Similarly to MongoDB, it does not support ACID-based transactions and therefore, it cannot replace relational databases in use cases where such transactions are required. Additionally, it cannot be used in use cases where ad-hoc querying is required (or where the query access patterns are not known in advance), due to the limitations of its query language; in such cases, MongoDB could be a better solution. Finally, due to the fact that the columns of a table must be defined using DDL commands, Cassandra is not the ideal solution for use cases that require very flexible schemas; in such cases, MongoDB could be used as well.

8.1.3 Amazon DynamoDB characteristics summary and suitable use cases

Amazon DynamoDB is a key-value store that provides very high, continuous availability, horizontal scalability, data durability and by default eventual consistency, although the latter can be configured to be strong. Tables in DynamoDB are almost schema-less (the only thing that has to be defined up-front is the primary key of the table), its query language is limited to simple operations that query only by the primary key and its secondary indexing support is very limited as well. Similarly to Cassandra, DynamoDB’s query language is limited in order to support and favour high availability, scalability and performance over complex ad-hoc queries. It is a DBaaS and therefore, database administrators do not need to worry about installing it and setting it up as everything is done automatically by Amazon; all they have to do is to specify the rate of read and write throughput they wish the database to achieve. DynamoDB can provide very high read and write performance; however, the higher the performance, the higher its cost will be.

DynamoDB is a very good choice in use cases where true, continuous availability is required and where the schema needs to be flexible. However, due to the simplicity and
limitations of its query language, it is not suitable for use cases that require more functionality than simple get/put operations. Being an elastic DBaaS of which the administrator can increase/decrease the performance at any time, it can be the ideal database to use for the initial release of online applications, when their usage pattern is not yet known.

An example of a use case where DynamoDB could be a good choice is in newly released online games, to manage the profile and the activity of the players. For a player’s profile and activity data management, simple get/put operations would suffice. Additionally, the AWS provisioned resources could be scaled up or down depending on the usage pattern of the game so as to minimise the losses or to maximise the profits.

Being a DBaaS that allows the owner to scale up or down the resource usage whenever they wish, DynamoDB could be a good choice in use cases where application owners do not have or do not want to invest money in hardware resources to host a local database. The free tier of DynamoDB can be used for newly released applications and therefore, application owners can have the opportunity to test an application before spending any money on hardware resources.

Finally, DynamoDB can be used in use cases where a pure key-value store would suffice to perform all the required operations. For example, a suitable use case could be the management of the shopping cart of a customer of an online retail shop. The only operations required to manage the shopping cart is simple get, put, delete and update operations.

Similarly to MongoDB and Cassandra, DynamoDB does not support ACID-based transactions; therefore, it cannot replace relational databases in use cases where such transactions are required. Due to fact that its query language is very limited, it should not be used in use cases where ad-hoc querying is required either; instead, MongoDB could be used in such cases. Finally, if the usage pattern for an application is known it might be better use a local database like MongoDB or Cassandra instead of DynamoDB, because the need of transferring data in and out of the AWS region will be eliminated, and because a local database may prove to be less costly than a DBaaS, particularly over long term.

### 8.2 Future work

For the purpose of this dissertation project, three NoSQL databases were evaluated; MongoDB, a document store, Cassandra, a wide-column store, and Amazon DynamoDB, a key value store. In subsequent evaluations a graph store (e.g., Neo4J) could be evaluated as well so as to complete the four main types of NoSQL databases. The three evaluated databases are amongst the most popular NoSQL databases as of today. However dozens of other NoSQL systems exist; in potential subsequent evaluations, more databases that support different functionalities can be evaluated as well. For example Redis, which is the most popular key-value store and it is an in-memory database and Apache HBase, which is the open-source implementation of Google’s BigTable.
In this evaluation, a set of quality attributes and performance metrics were selected to evaluate the databases. In subsequent evaluations additional quality attributes such as database reliability, robustness and security can be included so as to get a better understanding of the strengths and weaknesses of each database.

Finally, during the quantitative evaluation MongoDB and Cassandra were evaluated using a small number of nodes and a relatively small dataset (10 GB on 3 nodes – the usage of a larger dataset and cluster was investigated but did not take place due to time limitations). NoSQL databases were built out of necessity when relational databases could not handle big data efficiently any more, and one of their primary aims was to provide high horizontal scalability. For example, Cassandra was built to efficiently handle the inbox search functionality of Facebook and it is claimed that it can handle Petabytes of data with ease. Therefore, a more objective quantitative evaluation would be to benchmark the databases using big data workloads with size of data upwards of hundreds of Terabytes, and adequate hardware resources, clusters that would consist of hundreds of nodes.
Appendix A

MongoDB and Cassandra quantitative evaluation results

A.1 Data import performance

Throughput (operations per second)

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>3903.2</td>
<td>7613.25</td>
<td>14275.52</td>
<td>27806.74</td>
<td>50188.21</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>5618.09</td>
<td>13959.98</td>
<td>27403.52</td>
<td>43336.94</td>
<td>73846.15</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>2096.44</td>
<td>3997.6</td>
<td>8067.77</td>
<td>15857.28</td>
<td>25386.83</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>4868.55</td>
<td>10273.97</td>
<td>21436.23</td>
<td>41355.78</td>
<td>66898.95</td>
</tr>
</tbody>
</table>

Latency (milliseconds)

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>0.26</td>
<td>0.26</td>
<td>0.28</td>
<td>0.29</td>
<td>0.32</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>0.53</td>
<td>0.43</td>
<td>0.44</td>
<td>0.55</td>
<td>0.62</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>0.48</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.63</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>0.61</td>
<td>0.58</td>
<td>0.56</td>
<td>0.58</td>
<td>0.72</td>
</tr>
</tbody>
</table>
### A.2 Read performance

**Throughput (operations per second)**

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>60.9</td>
<td>68.96</td>
<td>90.09</td>
<td>108.41</td>
<td>126.22</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>173.61</td>
<td>236.41</td>
<td>341.97</td>
<td>453.68</td>
<td>494.34</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>62.03</td>
<td>63.49</td>
<td>81.2</td>
<td>96.72</td>
<td>105.26</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>199.07</td>
<td>282.75</td>
<td>355.98</td>
<td>458.01</td>
<td>504.2</td>
</tr>
</tbody>
</table>

**Latency (milliseconds)**

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>16.42</td>
<td>29.06</td>
<td>44.42</td>
<td>73.78</td>
<td>126.76</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>17.29</td>
<td>25.38</td>
<td>35.09</td>
<td>52.9</td>
<td>97.07</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>16.12</td>
<td>31.14</td>
<td>49.26</td>
<td>82.72</td>
<td>152.28</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>15.07</td>
<td>21.22</td>
<td>33.71</td>
<td>52.38</td>
<td>92.58</td>
</tr>
</tbody>
</table>
A.3 Write performance

**Throughput (operations per second)**

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>38.02</td>
<td>42.55</td>
<td>48.61</td>
<td>65.96</td>
<td>72.85</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>119.99</td>
<td>179.88</td>
<td>224.34</td>
<td>278.42</td>
<td>333.79</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>2040.81</td>
<td>4081.63</td>
<td>8779.63</td>
<td>17308.52</td>
<td>25449.34</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>5393.74</td>
<td>11003.12</td>
<td>23139.22</td>
<td>43462.51</td>
<td>67662.81</td>
</tr>
</tbody>
</table>

**Latency (milliseconds)**

<table>
<thead>
<tr>
<th>Threads per node</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB (1 node)</td>
<td>26.93</td>
<td>47.02</td>
<td>82.2</td>
<td>121.2</td>
<td>220.16</td>
</tr>
<tr>
<td>MongoDB (3 nodes)</td>
<td>25.01</td>
<td>33.36</td>
<td>53.49</td>
<td>86.23</td>
<td>143.86</td>
</tr>
<tr>
<td>Cassandra (1 node)</td>
<td>0.49</td>
<td>0.5</td>
<td>0.48</td>
<td>0.48</td>
<td>0.64</td>
</tr>
<tr>
<td>Cassandra (3 nodes)</td>
<td>0.56</td>
<td>0.54</td>
<td>0.51</td>
<td>0.55</td>
<td>0.71</td>
</tr>
</tbody>
</table>
Appendix B

Examples of query language commands in MongoDB, Cassandra and Amazon DynamoDB

B.1 MongoDB

B.1.1 Read

```javascript
db.tweets.find(
    { $and: [
        { lang: "en" },
        { retweet_count: { $gte: 50 } },
        { user.followers_count: { $gt: 1000 } }
    ],
    { "text": 1,
      "retweet_count": 1,
    }
)
```

This command uses the “and” logical operator, and the “greater than” and “greater than or equal” comparison operators to return the tweet text, and the total number of retweets of all the tweets that were written in English, they were retweeted 50 times or more, and whose author has more than 1000 followers. The attribute “user” of this example is an embedded document that contains (among others) the attribute “followers_count”.

B.2.2 Update

```javascript
db.tweets.updateOne(
    { _id: ObjectId("574c832e8ee9492e65466677") },
    { $set: { text: "Updated tweet text" } }
)
```
This command updates the document with the specified “_id” by using the “set” operator to set the tweet text to “Updated tweet text”. If the replaceOne command was used instead, the document would have been replaced with a new one with the same “_id” and with only one other field, “text”.

The update command has one restriction; once set, the “_id” field cannot be updated. Additionally, if no document matches the criteria specified in the <query filter> clause, a new document is created and imported into the collection. Finally, all updates are atomic to the document level.

B.2.3 Delete

    db.tweets.deleteOne( 
        { _id: ObjectId("574c832e8ee9492e65466677") } )

This command deletes the document with the specified “_id” from the database. All deletes are atomic to the document level as well.

B.2.4 Text search

    db.tweets.find( { $text: { $search: "United States\" } } )

If a text index exists on the “text” field of the documents, this command will return all the documents that contain the phrase “United States” in the tweet text.

B.2.5 Aggregation pipeline framework

    db.tweets.aggregate( 
        [ 
            { $project: { lang: 1 }},
            { $group: { _id: "$lang", total_number: { $sum:1 } }},
            { $sort: { total_number: -1 }},
            { $limit: 10 } 
        ]
    )

Initially, the “$project” operator is used to retrieve only the language attribute of every tweet and forward it down the pipeline. Consequently, the “$group” operator is used so as to group together the same languages. Additionally, in this stage, the total number of occurrences of each language is calculated and stored in the “total_number” field. Then, the results are sorted using the “$sort” operator and finally the top 10 results are returned using the “$limit” operator.
B.2.6 MapReduce

```javascript
db.tweets.mapReduce(
    function map(){
        emit (this.lang, 1);
    },
    function reduce(keyLanguage, valueOccurences){
        totalOccurences = 0;
        for (i=0; i<valueOccurences.length; i++)
            totalOccurences += valueOccurences[i];
        return(totalOccurences);
    },
    { query: { }, out: "results" }
)
```

The map function emits the “lang” attribute of the tweets as a key and the number 1 as a value. The reduce function receives the key-value pairs as input, it groups together the same keys and it calculates and returns the sum of the values of each key. Therefore, this operation counts the total number of occurrences of each language that was used in the imported data set’s tweets. Finally, the results are stored in the collection “results”.

B.2 Cassandra

B.2.1 Read

```sql
SELECT user_location_country, user_id, adults, children
FROM search_data
WHERE hotel_country = 1 AND user_location_country >= 10 AND user_location_country <= 15
ORDER BY user_location_country ASC
LIMIT 1000
```

This query returns the user ids as well as the number of adults and children for all the searches that were made by users who lived in countries with ids 10-15 and searched for hotels in the country with id 1. The results are sorted based on the user location country, and only the first 1000 rows are returned.

Due to all the restrictions imposed by CQL, data modelling in Cassandra follows a query-driven approach and a common practice is to maintain a separate table for each query. This results in data duplication across many different tables. For instance, the table schema that was specified in the data import process facilitates queries that use the hotel country and user location country in the WHERE clause. If another query...
required searching by hotel cluster, another table could have been designed to use the hotel cluster as the partition key. Based on the data retrieval requirements of the new query, the appropriate columns could have been duplicated in the new table as well.

B.2.2 Update

UPDATE search_data
SET adults = 2, children = 2
WHERE hotel_country = 23 AND user_location_country = 1
AND USER_ID = 213635 AND DATE_TIME = '2014-01-28 22:50:12'

This command sets the values of the columns “adults” and “children” to 2, for the row with the specified primary key.

B.2.3 Delete

DELETE
FROM search_data
WHERE hotel_country = 23 AND user_location_country >= 1
AND user_location_country <= 3

This command deletes all the rows from the search_data table that their hotel_country value is 23 and their user_location_country value is between 1 and 3.

B.2.4 Aggregate functions

SELECT MAX(children), MIN(children), AVG(children)
FROM search_data
WHERE hotel_country = 1 AND user_location_country = 7

This command calculates the maximum, the minimum, and the average number of children for all the searches that were made by users who live in the country with id 7, for hotels in country with id 1.

B.2.5 User Defined Functions (UDF) and User Defined Aggregate functions (UDA)

CREATE FUNCTION sumStateful (state tuple<int, int>, value int)
CALLED ON NULL
INPUT RETURNS tuple<int, int>
LANGUAGE java
AS
' if (value != null){
    state.setInt(0, state.getInt(0)+ 1);
}
state.setInt(1, state.getInt(1) + value.intValue());
}
return state,'

CREATE FUNCTION avgStateful (state tuple<int, int>)
CALLED ON NULL
INPUT RETURNS double
LANGUAGE java
AS
' return (double) state.getInt(1)/(double) state.getInt(0);'

CREATE AGGREGATE myAVG(int)
SFUNC sumState
STYPE tuple<int,int>
FINALFUNC avgState
INITCOND (0,0)

The first UDF, “sumStateful”, takes as input two variables, “state” (which is stateful) and “value”, of types tuple and int. Each time it is executed, it increments the first integer of the “state” variable by one, and it adds the “value” variable to the second integer. The second UDF, “avgStateful”, takes as input one variable, “state” of type tuple, and it divides its second value by its first, so as to calculate an average result. Both UDFs are called in the “myAVG” UDA. When the UDA is applied to a column of a table in the SELECT clause of a “read” query, it sets the initial values of the “state” variable to 0 (in the INITCOND clause) and consequently it executes the “sumStateful” UDF for each row. At the end, it executed the “avgStateful” UDF so as to return the average.

B.3 Amazon DynamoDB

B.3.1 Create

def put_item(
    TableName = 'reviews',
    Item={
        'id': -1,
        'ProductId': 'dummy',
        'UserId': 'dummy',
        'ProfileName': 'dummy',
        'HelpfulnessNumerator': -1,
        'HelpfulnessDenominator': -1,
        'Score': -1,
        'Time': 'today',
    })

This command imports a new review into the “reviews” table.

B.3.2 Read

get_item(
    TableName = ‘reviews’,
    Key = {
        ‘id’: -1
    }
)
This command retrieves the item with “id” equal to -1 from the table “reviews”.

B.3.3 Update

update_item(
    TableName = ‘reviews’,
    Key = {
        ‘id’: -1
    },
    UpdateExpression=‘set score =:i, summary=:s’,
    ExpressionAttributeValues={
        ‘:i’: -1,
        ‘:s’: ‘Updated’
    }
)
This command updates the values of attributes “score” and “summary” of the item with “id” equal to -1, to -1 and “updated” respectively.

B.3.4 Delete

delete_item(
    TableName = ‘reviews’,
    Key = {
        ‘id’: -1
    }
)
This command deletes the item with “id” equal to -1 from the table “reviews”.

B.3.5 Scan

scan(
    TableName = 'reviews',
    FilterExpression = 'Score > :sc',
    ExpressionAttributeValues = {
        'sc': {'N': 3}
    },
    ProjectionExpression = 'UserId, ProductId, Score'
)

This command reads every item in the table ‘reviews’ and returns the attributes “UserId”, “ProductId” and “Score” of all those items which had a score greater than 3.

B.3.6 Query

query(
    TableName = 'reviews',
    IndexName = 'score_per_product_index'
    KeyConditionExpression = 'ProductId = :pid AND Score >= :sc',
    ExpressionAttributeValues = {    ':pid': {'S': 'B000GG1OAI'},
                                    ':sc': { 'N': '4'}
    }
)

This command returns all the Items of the table “reviews” that their “ProductId” is equal to ‘B000GG1OAI’ and their score is greater than or equal to 4. This command uses the secondary index “score_per_product_index” to retrieve the results.

[1]
Bibliography


[15] Key-value stores [online]. Available at: http://www.aerospike.com/what-is-a-key-

[16] Key-value database [online]. Available at: https://en.wikipedia.org/wiki/Key-
value_database . Accessed: 10/05/2016

[17] Document databases [online]. Available at: https://ayende.com/blog/4459/that-no-


[19] Column family databases [online]. Available at: http://www.jakowicz.com/nosql-
 essentials-part-4-column-family-databases/ . Accessed: 10/05/2016


[21] Graph stores [online]. Available at: http://neo4j.com/developer/graph-


74


