PERFORMANCE ANALYSIS OF A DATABASE LAYER'S MIGRATION FROM RDBMS TO A NOSQL SOLUTION IN AMAZON AWS

MSc Thesis of Internet and Web Technology

written by

Carlo Butelli
(born June 26st, 1984 in Grosseto, Italy)

under the supervision of **Dhr. Dr. A.S.Z. Belloum**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Computer Science

at the *Universiteit van Amsterdam* and *Vrije Universiteit Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
Date of publication(soon)	Dr. Adam S.Z. Belloum
	Dr. Patricia Lago

VU VRIJE UNIVERSITEIT AMSTERDAM

UNIVERSITY OF AMSTERDAM

This is the Dedication.

Declaration of originality

Contents

List of Abbreviations

DBMS	DataBase Management System
NoSQL	"Not Only SQL"
RDBMS	Relational Database Management System
SQL	Structured Query Language
UML	Unified Modeling Language

Abstract

Nowadays, IT is dealing with a huge constant increase of data that is coming faster and in greater volumes than ever before. Companies are exploring ways to face the rise of these data and to create value out of them, being innovative and gaining competitive advantages. Such data is commonly referred to as *Big Data* and besides standard SQL databases, NoSQL databases have been created to better deal with this situation.

Traditional SQL databases provide powerful mechanisms to store and query structured data under strong consistency and transaction, guaranteeing data integrity and consistency. Nowadays, this kind of database turned out to be critical in managing the explosion of Big Data especially because they are not completely able to scale too big, too fast or with "diverse" data. This can reflect their "failure" to cope with high-volume, high-velocity, and high-variety of data.

This new technology called NoSQL database instead is gaining more and more popularity with the debut of social media and cloud computing. NoSQL basically deal with non-structured or semi-structured data and certain workloads seems to scale better and to be more cost-effective using this solution.

The main goal of this research is to investigate on the performance of this NoSQL database compared with an instance of the already popular SQL database underlying advantages and disadvantages by performing some benchmark analysis. The entire research has been done under the profile of a company that currently is intended to migrate the whole system to the cloud(Amazon AWS) and make use of the NoSQL Amazon DynamoDB for tables with high load of reads.

Chapter 1

Introduction

Nowadays, the huge growth in spread of mobile devices, social networks, “Internet scale” web applications and Web 2.0 are making the world more information-driven and companies have been starting to seek new ways to deal with the large amount of data coming up time by time. One of the major goal for companies is to get value out of those data, being innovative and gaining competitive advantages in order to increase the speed up of getting and tracking data in their own applications.

Today, software architects and developers have several choices for data storage and persistence. At the top of them there are two main technologies, the well known RDBMS and the novel data storage system called NoSQL, most commonly referred to as ’Not Only SQL’.

RDBMS guarantees and have reached an unmatched level of reliability, stability and support through decades of development. However, in recent years, the amount of useful data in some application areas has become so vast that it cannot be stored or processed by traditional database solutions.

Its structure presents some ”deficiency”. The information that are spread on several tables linked together through a clause called JOIN, eventually involves many query requests to the database producing high slowdowns. Moreover, those incoming requests are also outgoing, meaning that every simple *insert*, *update* or *delete* operation even involving a single record, it will entail a mix of requests to the database and joining several tables in a distributed system is pretty difficult end expensive.

On the other side, NoSQL is a term that describes a broad class of technologies surged in popularity providing a different approach to data storage compared with traditional SQL databases. Some refer to them as ’No relational’ or ’No RDBMS’, some others simple prefer to call them Distributed Database Management Systems. Among the main features they seem to be able to overtake some of the problems above by offering horizontal scalability, elasticity and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees. [13]

SQL in most typical situations scale vertically, which means that they can face the increase of loads by increasing RAM and SSD capacities, CPU and so on, on a single server.

In a Web application where the number of users and data will start growing beyond expectations making it hard to keep up with the data requirement, an RDBMS will start to feel the heat and a migration from a relational database to a non-relational one sometimes could be a powerful solution.

The popularity of NoSQL has some key factors as compared to the RDBMS systems. Some of them are the increase of data availability, a lightweight computing (no more operations on data aggregation), more scalability(from vertical to horizontal just adding Servers) even if it could present some drawbacks like data redundancy.

1.1 Scope of the Work

This thesis is aimed to independently investigate the performance of the NoSQL key-value store Amazon DynamoDB against Amazon RDS for MariaDB as SQL database in the provision of migrating the database layer from a RDBMS to a NoSQL solution. The comparison has been made through simple operations, that are, read, write, delete and update measuring their operational time to have a good estimation to what extent migrating the database layer (or a part of it) to a new solution could worth. This crucial step has been done by taking into consideration the literature review. An additional operation that iterates through all keys has also been investigated. Experimental results measure the timing of these operations and how the databases stuck up against each other.

1.2 Research Questions

RQ1 What are the main features of NoSQL databases and why companies are more and more going to adopt such structure for their database layer? Which trade offs are involved in the choice of a NoSQL database?

RQ2 Considering the increase of data available in a fast growing company, to what extent making use of a database layer based on NoSql instead of the traditional RDBMS may be useful?

RQ3 Which trade offs should the application owner make in order to face data integrity, data type, complex/ad-hoc queries also under the transactional point of view? DynamoDB for instance only support a few data types compared with SQL, which is the impact of such lack?

RQ4 ACID properties (Atomicity, Isolation, Durability and Consistency) are the golden rules which an ideal DBMS should follow and implement. In contrast, a NoSQL DB follows the CAP theorem (Consistency, Availability and Partition Tolerance). What is the impact that a NoSQL database can have on other application layers? And what happen in case that some functionalities are missing or have different non-functional properties? (Example of JOIN or other operations not supported from NoSQL)

1.3 Structure of the report

The project starts with a literature review introducing a background overview of both databases' landscapes to discuss their features, how their engines work and to recognize some strengths and weaknesses (and technologies used to deal with them). The purpose is also to explore the requirements typically posed to a NoSQL database systems and the techniques used to fulfill these requirements along with the trade-offs which have to be considered.

Because of the increasing number of available options about NoSQL solutions, and due to the fact that the company was migrating the system into Amazon Cloud Services, it has been decided to limit the scope of this work focusing only on the key-value store Amazon DynamoDB rather than other implementations like column-stores (such as BigTable), document stores (such as MongoDB), or some combination between a simple store and a more complicated store (like Cassandra, which is essentially a combination between BigTable and DynamoDB) [5, 6, 54].

The fourth chapter will introduce the framework used in support of all the operations mentioned above along with the design and the implementation in Django.

In conclusion, a performance benchmark will be performed towards both DBs and results will be compared also to get some proof of concepts. Throughout the experiment the Unified Modeling Language (UML) will be used to represent and underlining architectural and design differences for the chosen implementations.

1.4 Useful used tools

To perform all the benchmarks some tools have been used:

- **Amazon EC2 instance(t2.2xlarge)**. The instance is running *Ubuntu 16.04.1 LTS (Xenial Xerus)* and it is provided of **32***GB RAM*, **8** *Intel(R) Xeon(R) CPUs E5-2676 v3 at 2.40GHz* with **8** *cores*, high frequency and the ability to burst above the baseline. [18] This is the instance where Django application has been deployed to run all the benchmarks towards *Amazon RDS(MariaDB)* and *Amazon DynamoDB.*

- **Amazon RDS for MariaDB**. It is a managed relational database service. The instance provides *10.0.24-MariaDB Server*, *mysql Ver 14.14 Distrib 5.7.17, for Linux (x86_64)*, *InnoDB v. 5.6.28-76.1* storage engine, **500**GB storage (up to 6TB). [46]

- **Amazon DynamoDB**. It is a fully managed fast and flexible NoSQL database service that supports both document and key-value store models. Data is stored on solid state disks (SSDs) and automatically replicated across multiple Availability Zones in an AWS region, providing built-in high availability and data durability. [45]

- **Django Framework v. 1.9.4**. Django is a high-level, free and open source Python Web framework. It provides to the developers a way to handle user authentication (signing up, signing in, signing out), a management panel for your application, forms, models to query the databases, a way to upload files and so on. [35]

- **Driver boto3**. The Amazon Web Services (AWS) SDK for Python. It allows Python developers to write software making use of Amazon services(S3, EC2, DynamoDB, ...). Boto3 provides an easy to use, object-oriented API as well as low-level direct service access. [19]

Chapter 2

Background and Literature Review

This chapter provides a brief introduction to the Relational Database Management Systems (RDBMSs) with their properties and an overview of NoSQL databases presenting some data models and their features. It worths to say that traditional RDBMS does not have anything wrong, just they were not built to deal with a really huge amount of data and the entrance in the market of NoSQL databases with their roughly cheap hardware needs, has designed the necessity of some SQL databases to be converted to NoSQL ones. [12] However, the real goal of NoSQL is not to reject SQL but instead to be used as alternative data model in situation where relational databases do not work well enough.

2.1 Related Work

In [10] A. Diomin and K. Grigorchuk show some useful key criteria to take into account when the choice of a DB is required.
Scalability has to be guaranteed since when your application's traffic rises, rapid scalability upon requests is strictly required to fill up the lack of your DB's capacity. In the same line, when the system is idle, it has to be able to reduce the used resources. NoSQL DBs provides horizontal scalability by splitting the system into smaller components hosted on several physical machines that can be added on the fly and performs the same operation in a parallel manner.
Performance is important to persistently provide low latency regardless data size or task. *"In general, the read and write latency of NoSQL databases is very low because data is shared across all nodes in a cluster while the application's working set is in memory."* [10]
High Availability is necessary because large amount of money could be lost from a business at any time the application is down. It is strongly required the possibility to provide disaster recovery, to perform online upgrades, backups or easily remove a node for maintenance without affecting the availability of the

cluster.
Ease of deployment since SQL databases own a rigid schema, if the application change, most probably the DB's schema will have to change too while NoSQL DBs are know to be schemaless so this constraint should not be an issue anymore. [44]

In early times there has been many papers studying relationships between NoSQL and relational databases providing overviews and showing their characteristics, benchmarks and performance. [8, 59]

From [11] NoSQL seems to have a kind of structure that allow to store and process large amounts of data of (almost)any type providing high availability and being way faster than RDBMS.

Besides all the advantages like speed, better performance and high scalability NoSQL may bring into the market, *Leavitt* in [49] shows that they still present some drawbacks when compared with relational databases. He especially observed that even though they are fast in performing simple operations, they becomes tedious when operations get more complex.

Nowadays, security has becoming more and more an undertaken feature being on of the most considerable concern for IT Enterprise Infrastructures. Unfortunately, the weakness of security in NoSQL is mainly due to the fact that their designer focuses on other purposes than security and Authentication/Encryption where implemented are almost absent. [15]

The research made by *Indrawan-Santiago* in [47] had the intention to gather together some basic comparisons that could be made among NoSQL DBs and against relational databases. The research that eventually shown the complementarity between NoSQL and RDBMS included transaction and data model, indexing, sharding, support for ad-hoc-queries and license type.

In [27] a comparison between relational and NoSQL databases had been described, precisely by using Oracle and MongoDB. From the study came up that the query time calculated performing an insert operation was a factor higher in Oracle than in MongoDB while performing delete and updates took several factors more than MongoDB.

2.2 Overview of RDBMS

A database can be considered as an organized collection of data [26], a huge library where every shelve is represented by a table containing records, split by attributes (columns). The system which is in charge to manage all the interaction with the database is called Database Management System(DBMS) and it is represented by an organized set of facilities to controls the organizations, accesses, retrieval of data and to maintain one or more storages. [26] DBMSs can be classified based on their criteria in Data Model, User Numbers and Database Distribution.

- **Data Model** comprehends several types:
  - **Relational Model(RDBMS)**: nowadays the most widely used data

  model. It uses structures consistent with the *first-order predicate logic* based on the relational model proposed by E. F. Codd in 1970 [28] and the one it has been used in this work. [4]

  – **Hierarchical Model**: data is organized in a rooted-tree-like structure (no cycles) and stored as *records* linked together by *links*. The relationships between a parent and a child must be one-to-many or one-to-one and the database schema is represented as a collection of tree-structure diagrams where every diagram has one single instance of a database tree rooted at a dummy node. This model's structure was the base of one of the firsts database models used and implemented in IBM's Information Management System(IMS). [3]

  – **Network Model**: it can be seen as a generalization of the hierarchical model where each data object can have multiple parents and each parent object can have multiple children forming a *lattice-type* structure in contrast to the *tree-like structure* of the hierarchical model. It is useful to represents real-world data relationship more naturally and under less constrained environment. [24]

  – **Object-Relational Model(OODBMS)**: it has been introduced in recent years and it is one of the newer and more powerful models of all the aboves. Information are represented by objects and it adds object-oriented concepts to facilitate data modeling and its relationships. [2]

- **User Numbers**: it can be distinguished in DBMS which supports one user at a time(single-user), or in a DBMS, which supports multiple users concurrently(multiuser). [63]

- **Database Distribution**: it in turn is divided in four main distributions:

  – **Centralized systems**: the database and DBMS are both placed in the same site used from other systems too.

  – **Distributed database system**: the database and DBMS are placed in different locations each one connected to each by a computer network.

  – **Homogeneous distributed database systems**: make use of the same DBMS from multiple locations.

  – **Heterogeneous distributed database systems**: it makes available additional common software in order to support data exchange among locations that make use of different DBMS software.

Further and in-depth description of these models is beyond the scope of the present work. However, the next section will show a bit more in details the Relational Model that is on protagonist of this research.

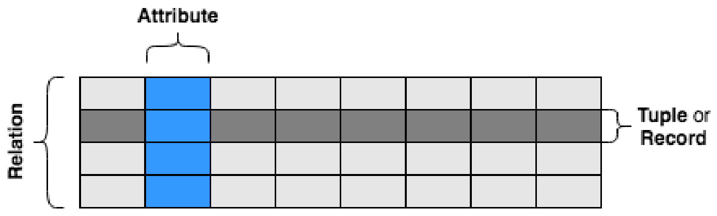


Figure 2.1: Example of a relational model

2.2.1 The Relational model

Over the years the relational database management systems (RDBMS) have been the leading technology in the IT world. The relational model is fundamentally based on two mathematical components: *first-order predicate logic* [4] and the *theory of relations*. This approach of using the relational systems in all scenarios came under scrutiny with the introduction of the web. The relational data model works well with the traditional applications where the data is not massive and distributed. Despite being on top for several years the capability of the RDBMS for processing and handling large amount of data remains in question.

The relational model, as discussed in [28] was developed to assign some issues like removing data consistency, hiding from the users the organization of the data in a machine and to protect users' activities (and applications' programs) from growing data types and internal data changes. The data structure in a relational model is made by several components identified as relation, entity, domain, attribute(column), tuple, and attribute's value. As in fig. 2.1 a relation(table) is represented by the Cartesian product of a set of records(rows or tuples) identified by a name and is then intended to represent data through tables composed by rows and columns. A schema is a description of the data in terms of data model.

Basically, a database is represented by a collection of one or more *relations*, where each one is based on a *relation schema* and a *relation instance*. The *relation schema* is used to specify relations' name, attributes' names A_i and attributes' domains(types) D_i while the *relation instance* is a finite set of tuples. [4]

In a *relation schema* denoted by $R(A_1{:}D_1, A_2{:}D_2, \dots, A_n{:}D_n)$, R stands for the relation name while $A_1, A_2, \dots, A_n$ represents a list of attributes and each D_i indicates the domain of the related attribute(column).

Eventually, a **Relational Database** is a collection of relations with distinct relation names and **Relational Database Schema** is the collection of relations' schemas in the database and **Integrity Constraints**.

In an abstract level, tables represent entities like the ones in fig. 2.2 that are *user* and *food* while *user_favorites* represents the relation between these two entities. The relational style is very useful to design a database schema mapping real world objects as database tables and relations among them expressed by

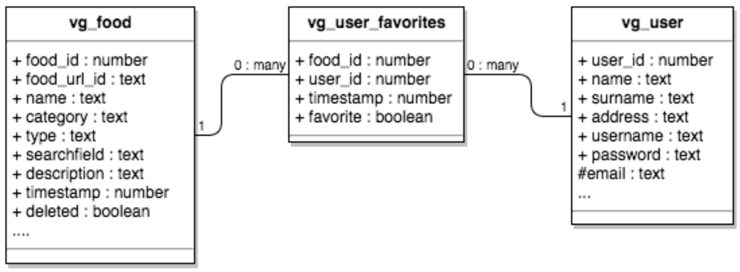


Figure 2.2: Example of a database schema

primary and foreign key to link tables together. [4]

2.2.1.1 Integrity Constraints: the concepts of Primary key and Foreign key

A *primary key* is represented by a single column or a combination of columns(in this case is called *composite key*) and it identifies uniquely a tuple in a relation. In a *composite key*, every column taking part to the key, should be a foreign key, but not necessarily [31]. If more attributes could identify the tuples, such attributes are called *candidate keys*.

A *foreign key* is again a column or a set of columns in one relation that make a reference to the primary key in another table. As example, in fig. 2.2 the table *vg_user_favorites* maintains two *foreign keys* that are *food_id* and *user_id*.

Every table must satisfy several constraints in order to be a valid relation. The following main classes of integrity constraints are described in [26, 31]:

- **Entity Integrity Constraint**: every *primary key* has to be unique and not to contain NULL values since they are used to identify every tuple in a relation.
- **Referential Integrity Constraint** states that if a relation has a *foreign key*, this one must refer to an existing tuple. It is intended for maintaining consistency between tuples of two tables. Records from a table, for instance, cannot be deleted/changed if matching records exist in another table.
- **Domain Constraint** designates and important condition to be satisfied, *"every attribute value inserted in a specific column must be in line with the domain specified for such column"*.

Other types of constraint are extensively explained in [26] section 5.3.∗.

2.2.1.2 The concept of Normalization

One of the main goals of relational database design is to normalize a relation schema together with a set of data dependencies into an suitable normal form [31] to eliminate issues related to inconsistency and redundancy improving data integrity. E. F. Codd introduced the term normalization and what is well known as *First Normal Form*(**1NF**) that consist of eliminating repeating data by creating a new table for each group of related data identified by a primary key. [28] Later on he also defined the *Second Normal Form*(2NF) and *Third Normal Form*(3NF) [29] that are presented as the following:

- **2NF**: if the table has some values which are the same for multiple records then move them into a new table and link the two tables through a foreign key.
- **3NF**: keep in the table only fields that depend on the primary key, at maximum put non depending fields into another table.

A few years later together E. F. Codd and Raymond F. Boyce introduced a form that looks like slightly strong than 3NF called ***Boyce-Codd Normal Form***. [30] This form ensures that no redundancy can be detected using functional dependency [34] and every candidate keys do not have any partial dependency on other attributes.

All these normal forms are someway related: every relation in BCNF is also in 3NF which in turn is in 2NF and every relation in 2NF is in 1NF.

2.2.2 The ACID properties

A transaction is a very small unit and it may contain many other low-level tasks. RDBMSs ensure four properties under the name of **ACID properties** in order to guarantee completeness, data integrity and to get transaction reliability in concurrent accesses and system failures. ACID stands for [4, 26]:

- ⋆ **Atomicity**: based on "all or nothing" statement. Every transaction must be considered as an atomic unit meaning that either all or none operations are accomplished.
- ⋆ **Consistency**: every completed transaction brings the database from a consistent state to another so any values' changes in an instance are consistent with other values' changes in the same instance.
- ⋆ **Isolation**: every transaction is isolated, so in case of concurrent transactions, the system is able to prevent conflicts and it will eventually end in the same state as if the transactions would have been performed serially.
- ⋆ **Durability**: every changes applied to the database once a transaction has been committed, it will persist in the database in any case of power failures, errors and crashes.

RDBMs achieve atomicity and durability by a log file that keeps track of any updates performed by any transaction so that if a error/failure occurs the database rolls back the transaction itself unbinding the update. Using a log file gives the advantage that nothing in memory has to be maintained and in case of failure the rollback can be performed when the system restarts. If a transaction succeeds but the crash comes up right before the data was saved to the disk then all the updates can be started over through the log file.

Isolation is usually obtained through "locking". Locks can be shared by any number of transactions that only want to read the data but not to update it(or vice versa). If a transaction is read-locked then no write-locks can be performed on the same transaction. If the transaction is write-locked no other write/read locks can do anything until the lock has been released.

2.2.3 SQL Language

SQL, also known under the name of Structured Query Language, is a comprehensive query language based on Codd's concepts built from a simplified version of DSL/Alpha that was previously developed by IBM [25].

It represents a combination of **Data Definition Language(DDL**, *used to define schemas*), **View Definition Language(VDL**, *used to specify user views and their mappings to the conceptual schema*) and **Data Manipulation Language(DML**,*used to define set of operations to manipulate data*) and it mainly allows to perform query language to define both conceptual and external schema, user and application views as results of predefined queries. [34]

Relational DBMSs make totally use of it to query data in databases' relations. The following example, based on the relations in fig.2.2, shows how simple is to query a table with SQL:

```
SELECT *
  FROM vg_user U
  WHERE U.surname = "Rossi"
```

This example allows to retrieve all the users with surname equal to "Rossi" from the relation *vg_user*.

2.2.4 Amazon RDS for MariaDB

Amazon RDS(Relational Database Service) is a solution that requires no dealing with administration and maintenance since RDS is born to be a fully functional alternative to common hardware databases. It is fast, scalable and can be replicated among Availability Zones(AZs) in order to get a better level of accessibility because low-latency network connectivity is provided towards the other Availability Zones in the same region. As well, your applications can be protected from failure in a single location if just they are launched from separated Availability Zones. RDS supports the management of several well know database engines like MySQL, PostgreSQL, Amazon Aurora, MariaDB, Microsoft SQL Server and Oracle. [46]

The basic "building block" of Amazon RDS is the *DB instance* which represent an isolated database environment in Amazon cloud. Each DB instance can own from 5GB to 6TB of associated storage capacity. Storage capacity comes int three different ways that also have distinct performances and costs: Magnetic, General Purpose (SSD), and Provisioned IOPS (SSD). [46]

Moreover, each DB instance can be launched in various Availability Zones and Amazon accordingly provisions a *"synchronous standby replica of your DB instance in a different Availability Zone"* to minimize latency in case of system backups, to supply data redundancy, fail-over support and so on. [46]

More detailed and fully documented information about Amazon RDS can be found in [46].

2.3 Sharding

Every application's developer or DB's administrator is aware that if the load of transactions and DB's size increase linearly, the response times rise logarithmically. Performance and scalability mainly depends on how database management systems are built and in turn they rely on Memory, CPUs and Disks.

Of course, it is not enough increase only one of these components to reach performance improvements or scalability but if more processing cores are provided also memory and hard drive capacity need to be increased.

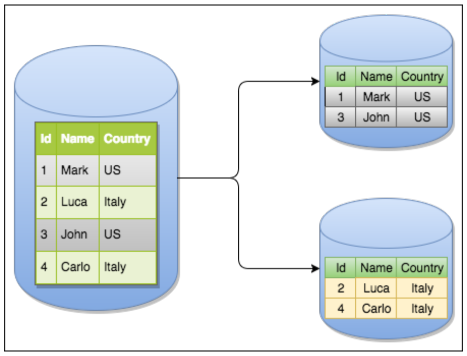


Figure 2.3: Example of Sharding

Due to the way they are structured, relational databases usually scale vertically, a single server hosts the entire DB ensuring acceptable performance for cross-table joins and transactions. This places limits on scalability, gets expensive and adds a few failure points for DB infrastructure.

The term *Sharding*(or partitioning) represents the approach under which data can be stored over many thousands of computers to improve the throughput and the overall performance especially if related to high-transactions.

Instead of storing data in a DB on one single machine, it is partitioned in smaller DBs called *Shards* across a cluster of nodes allowing to structure the DB with smaller index size, smaller working set and improving the handle of writing operations. Nodes can be added to the cluster to increase both capacity and write/read operation performance, contrary, if the demands decrease the size of a sharded DB cluster can be reduced. This ability goes under the name of *Horizontal scalability*.

Relational DBs do not provide the ability to scale horizontally but Sharding could be achieved through Storage Area Networks(SANs) and other complex arrangements for making hardware act as a single server.

The concept of *Database Sharding* has been gaining popularity over the past several years, due to the high increase in transaction volume and size of business application DBs.

However, a couple of drawbacks can be found in relational database with the use of operations like *Joins*, used to combine associated tuples from two relations into single ones [34]. It is not difficult to realize that, as joins work with two relations, performing such operation in a sharded DB would require many search requests to all the nodes containing items from one of the two relations causing high network traffic. Moreover, making use of sharded DB, the probability of node (or connection) failures increases hand in hand with the increase of the number of used nodes. Relational databases are not well suited to scale in this fashion, they look more appropriate towards vertical scaling (e.g. adding more memory, storage, ...) but of course it rises the limit of the number of resources that eventually can fit inside a machine. There could come a point where horizontal scaling becomes the only option.

2.4 Replication

Regardless the evident advantage of added performance DB, replication is recommended over sharding as it provides redundancy and increases data availability. Having multiple copies of data in different DB servers allows not only to protect the DB from loss on a single servers but also to recover from hardware failures or service interruptions and to increase the locality and availability of data in distributed applications. With additional replicas of the data, each one can be dedicated to different tasks like disaster recovery, reporting, or backup. e.g with more and more page requests, more and more reads are forwarded and replication may be used to increase read capacity, replicate the DB to read servers in such a way that the load balancer can forward writes operations to master and reads to the read server.

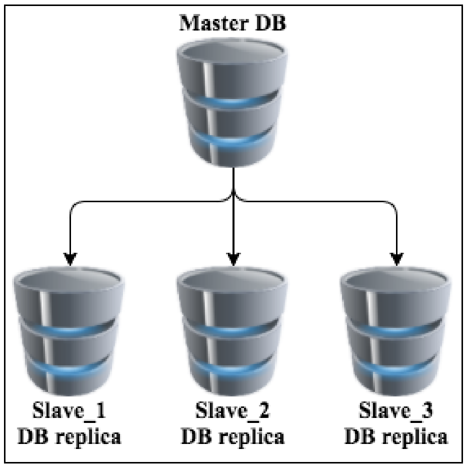


Figure 2.4: Example of Replication

Is common use to replicate database to different locations(or regions) in order to decrease latency in requests since data will be closer to the user. In any case, write operations are the drawback in replicated database environments because every write has to be forwarded to each node supposed to store such data item. [1] Depending on the desired level of availability and consistency a DB requires, there are two recommended ways to perform such operation:

- Commit the write towards all the nodes.

- Commit it towards one or a limited number of nodes then send it asynchronously to all the others.

2.5 Eventual & Strong consistency

In talking about *consistency* the scenario where every entity owns a copy of the data object is take into consideration. Conflicts can arise because each node can perform updates on its own copy and users connected to different nodes may see different values.

- **Eventual Consistency**: (also called *weak consistency*) each node communicates to each other its own changes and eventually after a fixed period of time without any other value's modification they agree on a definitive value. The update operation here ends as soon as the local copy of the item is updated then it is "broadcasted" to all the other nodes. The system does not guarantee that at every access the most updated value will be picked up during such period of time called *inconsistency window*. Moreover, in case of data replication, during the *inconsistency window* the process that writes the values could also get a non updated version of the data. [62]
- **Strong Consistency**: when a user makes an update to the data the system does not return anything instantaneously indeed it locks all the accesses to any copy of the data until all nodes have agreed on such updated value. Doing so every user will always read the same version of the data at the same time. [41]

2.6 NoSQL Databases

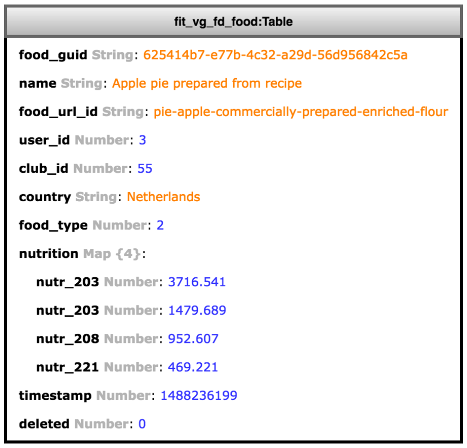
There are a lot of NoSQL database solution to take into account and it is hard to keep track about where they shine, where they fail or even where they are different, as implementation details change quickly and feature sets evolve over time. [10] A common feature is their capability to store big data and work with Cloud computing systems. Many of them are able to offer horizontal scalability and provide higher availability than relational databases by sacrificing querying capabilities and consistency guarantees. Every outstanding database has been designed for a specific class of application or in order to achieve a precise solution based on clear-cut system properties' requirements. The reason behind the presence of such large number in different database systems would be pretty clear thinking that any system would struggle to achieve all desirable properties at once.

2.6.1 NoSQL data models

The most common way to make a distinction among NoSQL databases is based on how they store and allow access to data. In the following subsections five different data-models' categorizations of NoSQL DBs will be presented and briefly described [47].

2.6.1.1 Key-value store

Key-value stores are the simplest among the NoSQL category and consist of a set of key-value pairs with unique keys. They work similarly to a traditional dictionary or hash table(where values are retrieved by using the key) but distributing keys and values among a set of physical nodes.



```
fit_vg_fd_food:Table
food_guid String: 625414b7-e77b-4c32-a29d-56d956842c5a
name String: Apple pie prepared from recipe
food_url_id String: pie-apple-commercially-prepared-enriched-flour
user_id Number: 3
club_id Number: 55
country String: Netherlands
food_type Number: 2
nutrition Map {4}:
   nutr_203 Number: 3716.541
   nutr_203 Number: 1479.689
   nutr_208 Number: 952.607
   nutr_221 Number: 469.221
timestamp Number: 1488236199
deleted Number: 0
```

Figure 2.5: Example of key-value store

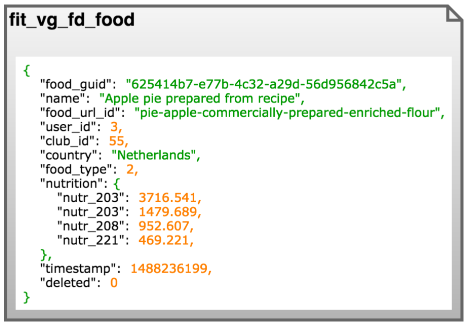
The data is stored in two parts, the real data that represents the value stored(string, integer, list, dictionary and so on) and a string representing the key to reference that value. Their very simple abstraction makes the database easy for partitioning and querying the data reaching in general low latency and high throughput.

However, due to their simplicity, Key-value stores just support basic **CRUD** (Create, Read, Update and Delete) and conditional operations. They are not suitable for applications with complex query operations like join or range queries. Due to this drawback, often data has to be analyzed in the application code to extract information, where queries more complex than simple lookups are required.

Key-value stores turns out to be in general good solutions if the application presents only one kind of object and the main operations are look ups of objects based on one attribute. Examples of widely used Key-value stores are DynamoDB [45], Project Voldemort [51] and Redis [60].

2.6.1.2 Document store

A Document-oriented DBs use a document oriented model extending the basic key-value store concept and storing data in the fashion of semi-structured formats such as JSON, XML or PDF [9]. Documents are addressed by using a key(simple string or a string that refers to a path or URI) that represent a sort of ID and uniquely identifies a document. To some extent every docu-



```
fit_vg_fd_food
{
  "food_guid":  "625414b7-e77b-4c32-a29d-56d956842c5a",
  "name":  "Apple pie prepared from recipe",
  "food_url_id":  "pie-apple-commercially-prepared-enriched-flour",
  "user_id":  3,
  "club_id":  55,
  "country":  "Netherlands",
  "food_type":  2,
  "nutrition": {
    "nutr_203":  3716.541,
    "nutr_203":  1479.689,
    "nutr_208":  952.607,
    "nutr_221":  469.221,
  },
  "timestamp":  1488236199,
  "deleted":  0
}
```

Figure 2.6: Example of a document

ment looks similar to a record in relational databases but way more flexible because they are schema less. While every record in a relational database may contain data in each field or it may leave unused field empty but still there, a document store is schema less and so every document may contain different data as different sized fields, they may be similar to each other document but do not have to be the same. They also allow to wrap key-value pairs in a document and due to this their complexity becomes slightly higher than key-value stores. [9] It is not advised to make use of a document-store if the data own a lot of normalization and relations [9], but, if the data does not need to be stored in uniform size or the domain model can be partitioned in some parts then a document store looks the right choice. MongoDB [54] and Apache CouchDB [40] are two of the most known and used document-stores nowadays.

2.6.1.3 Graphs database

Graph databases use the graph theory approach to store entities and relationship among these entities. The graph is made by two components, nodes, representing the objects and, edges, representing the relationships between those objects(interlinked elements). Nodes also have properties and are organized by relationships allowing to find interesting patterns between the nodes. To optimize the lookups Graph DBs use a so called *index free adjacency* technique where every node has a direct pointer to the adjacent node. With this technique millions of records can be traversed. However, it is very difficult to achieve sharding because it is very hard to cluster them. Graph databases are widely used in applications like social networks, bioinformatics, security and access control, content management etc. Neo4j [55] and Titan DB [43] are some of the notable Database-as-a-Service(DBaaS) provider using graph data stores.

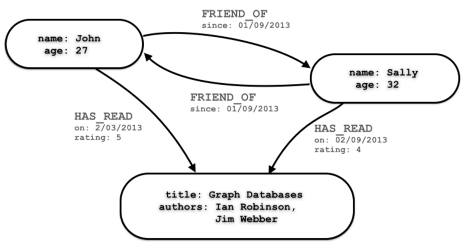


Figure 2.7: Example of Graph data structure[1]

2.6.1.4 Wide-column store

The main feature of Wide-column stores is that instead of storing data by rows(as it happens with relational databases), they store data in column families that are nothing more than groups of related data often accessed together, in this manner, data can be aggregated rapidly with less I/O activity.

This kind of databases look like a distributed multidimensional sorted map where columns are defined for every row (key-value pairs) in such a way that

[1]Thanks to `https://s3.amazonaws.com/dev.assets.neo4j.com/wp-content/uploads/data-modeling-5.png`

any row can have different columns and any number column can be added to any row. [9]

To make things more clear, as shown in fig.2.8, a column family could be seen as a container of rows(in RDBMS) everyone consisting of multiple columns and everyone identified by a key. This structure offers flexibility in data definition and allows to apply data compression algorithms per column.

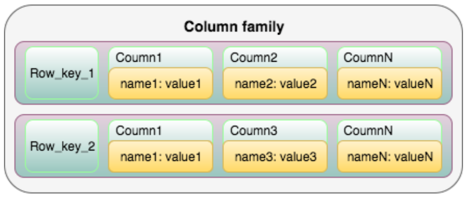


Figure 2.8: Simple example of Column-store

The storage scheme is identified by both *row keys* to identify rows and *column keys* which in turn identifies columns [14]. On disk data values are stored from the same column family and from the same row in lexicographic order of their keys in such a way that they result physically co-located. However, to retrieve a row, a single lookup is not enough instead, a join from the columns of all the column families is required.

Examples of well known Wide-column store are Google Big Table [5] and Cassandra [39].

2.6.2 The CAP Theorem

In contrast to the ACID transaction properties respected by relational databases, NoSQL databases trusts on the **CAP theorem** exposed in 2000 during a Symposium on Principles of Distributed Computing. The theorem presented by Eric Brewer and later confirmed by Gilbert and Lynch in [42] brought to the light an upper bound based on a trade-off between:

- ***Availability***(A): data must always be available,
- ***Consistency***(C): data is always the same no matter which replication or server,
- ***Partition Tolerance***(P): database is able to work even in presence of machine/network failures.

According to this theorem, in distributed systems, only two of the three properties could really be guaranteed at a specific moment in time [16]. Brewer stated that in partitioned system having both consistency and availability was not fully possible because in case of a host's failure that will lose the connection with the other nodes, for instance, it will have to choose if to preserve availability(AP) keep processing requests form clients and so violating consistency or to guarantee consistency rejecting clients' requests and sacrificing system's availability(CP). Another scenario is the one with a single-node system where availability and consistency can be preserved at the price of lacking completely in partition tolerance(CA) [16]. Depending on the use cases, the most important combination needs to be chosen. When the consistency of data is crucial, relational DBs should be used but when data is distributed across multiple servers, such consistency becomes hard to achieve.

Cap theorem has then evolved into what is now known as **BASE** principle which means *Basically Available*, *Soft State* and *Eventual Consistency* and that is characterized by high availability of data, while sacrificing its consistency [56, 57].

2.6.3 Query language

Despite relational DBs that use the SQL language to make queries, NoSQL DBs do not have any real standard query language. Most of the NoSQL DBs have created their own query language, some examples are MongoDB which uses mongo query language , Cassandra that make use of CQL(Cassandra Query Language) or boto3 used by DynamoDB and S3.

2.6.4 Overview of Amazon DynamoDB

DynamoDB is the high-performance, self managed, NoSQL database service solution build for the cloud and presented by Amazon [17].

Self-managed means that the service takes care about all the building blocks required to make a database scalable, the management of the database software and the provisioning of hardware needed to run it, allowing developers to focus on building applications rather than managing infrastructure. [45] The only instruction needed is telling the service how many requests it has to handle per seconds and it does the rest automatically. If the application takes off and it has to be able to handle thousands of thousands of requests, just rise the provisioned throughput and automatically the data will be spread towards a sufficient number of nodes to provides consistent performance and protect them against downtimes. [45]

Amazon DynamoDB provides two consistency options:

- *Eventual consistency* which maximizes the throughput at the expense of not having the last updated data.
- *Strong consistency* which reflects all write and updates.

Based on these two options two kind of secondary indexes can be crated: *Local Secondary Indexes*(**LSI**) that supports both strong and eventually consistent read/write options and *Global Secondary Indexes*(**GSI**) which only supports eventual consistency. As most of the key-values store

To guarantee fast access to the data, instead of using traditional hard-drives to store data, DynamoDB make use of Solid-state drives(SSD). In addition of being fast, DynamoDB is also extremely reliable and to enforce high availability, read consistency and durability DynamoDB applies replication of each table across three geographically distributed AWS Availability Zones creating a backup copy of the table in one or more geographic locations. [45]

DynamoDB does not have any limits on data storage per user, nor any maximum throughput per table.

Chapter 3

Framework design and DB engines

Application developers surely have plenty of experience in the use of relational databases and SQL. Besides underling all the differences, working with NoSQL databases brings to discover that eventually they have even more similarities than it is thought.

The design and architecture of data models are key factors in the flexibility of any database considering that they determine how data is stored, organized and can be manipulated. Understanding differences and similarities between NoSQL and RDMS database engines, their data models and how they work might help to better interpret the results obtained from our benchmarking analysis. For instance, it could be interesting to find out that the way a DB stores its data could impact the query time needed to retrieve such data in respect to the other database. For this purpose, this chapter aims to describe the data models used by Amazon RDS for MariaDB and Amazon DynamoDB comparing SQL basic statements with their equivalent NoSQL operations.

3.1 How do DBs store data?

In relational databases every information is supplied through models explained in section (2.1). By using an user-oriented approach, SQL queries work out pretty well assuming that data would be aggregated in one place to give all the needed information at once. However, creating the DB schema providing transactional guarantees and maintaining this kind of pattern requires a lot of time and effort.

3.1.1 Amazon RDS for MariaDB

The basic building blocks for Amazon RDS are represented from the so called DB instances. Each instance is an isolated database environments in the cloud, running its own DB engine with its features and supporting from 5GB up to 6TB of associated storage capacity [46]. Amazon RDS supports various DB engines like MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft SQL Server. However, this work has been focused only on MariaDB to represent the relational database category.

The data-structure in DBMS varies considerably, from fixed/variable length records of structured data to variable length records of opaque data, hash table entries and nodes of tree indexes(e.g. B-Tree, R-Tree, ...). Most commonly RDBMSs use variations of B-Tree as organizational structure for information storage to separate user applications from the physical database. Some examples could be found with MySQL which uses *B+ tree* to store data indexes, *InnoDB* stores both data and index file into the memory while *MyISAM* maintains only the index file, MariaDB instead uses a fork of InnoDB called *Percona XtraDB* which incorporates InnoDB's ACID-compliant design and MVCC architecture building greater scalability and providing a good tuning degree. [52]

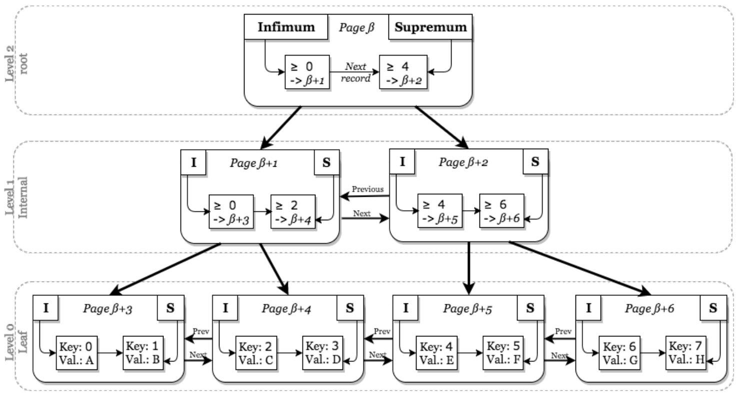


Figure 3.1: Example of B-tree structure

B-Trees [7] owns a particular structure well suited to store large sorted dictionaries on a fixed size blocks called pages(fig. 3.2). In each page, *Infimum* and *Supremum* point respectively to the lowest key value and the highest key value inside the page itself. Every page is linked with the previous and the next pointers in ascending order by key while each record is only linked to the next one again in ascending order by key. Beside the *Leaf Level 0*, fig. 3.1 shows that there are other two levels, respectively *Internal Level 1* and *Root Level 2*. Both internal and root level point to the lower key of the child and maintain the *node pointer* which is represented by the child page number.

Pages are in general maintained in a storage device such as HDD or SSD. In order not to lose any required data, it is advised to adjust each page size according to the device blocking factor, since that, when a request is made, the disk always read/write an entire block at a time even if the data requested is not large.

Supposing to have a disk block size of 6KB and we want to forward a read

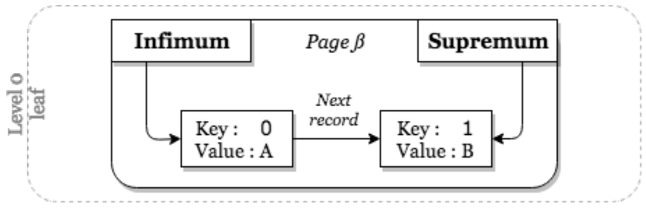


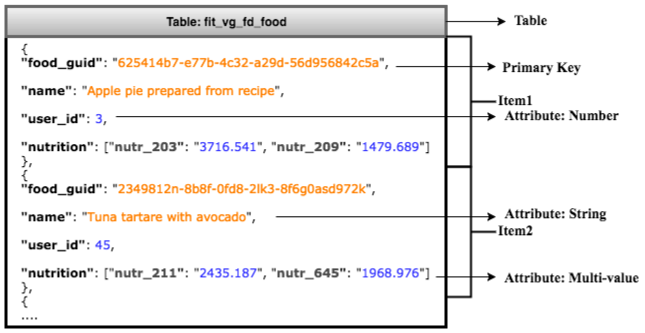
Figure 3.2: Example of B-tree page

operation on a sequence of 150 bytes of data. The disk firstly reads the whole 6KB block, containing the 150 byes, into its cache, the 6KB of data is then copied into the file system's cache and finally, only the 150 bytes requested will be copied into the program's buffer. In this case, it would have been truly inefficient to have pages of 1KB or 4KB, since the disk would have been reading 6KB regardless.

Moreover, to reduce the impact of I/O requests, the database system memory maintains an index of cache pages so that firstly DBMS performs a lookup of such index to determine if the page containing the requested data is in the cache, or not. If yes, a file I/O is avoided and even if also cache index lookup takes time, it is way less expensive than direct I/O request toward the database.

3.1.2 Amazon DynamoDB

DynamoDB data model is made by three main components: tables, items and attributes. Tables are similar to the ones used in relational databases with the difference that in DynamoDB they do not follow any precise schema, those only need a defined primary key and a specified data type to uniquely identify every item in a table and by then, any number of items can be stored in a single table(no other attributes are defined in advance). The primary key in DynamoDB can be created in two ways: by a simple single attribute, called *Partition key*, or, by two attributes forming a *Composite key*(*Partition key* and *Sort key*). Items represent records composed by a primary/composite key and an arbitrary number of attributes that can also be defined at runtime. The only limit is on the item's size which cannot exceed 400KB. An attribute is composed by a name and a value or set of values. Also here, there is no limits in size for individual attributes but still the total item's value cannot exceed 400KB.



```
{
"food_guid": "625414b7-e77b-4c32-a29d-56d956842c5a",

"name": "Apple pie prepared from recipe",

"user_id": 3,

"nutrition": ["nutr_203": "3716.541", "nutr_209": "1479.689"]
},
{
"food_guid": "2349812n-8b8f-0fd8-2lk3-8f6g0asd972k",

"name": "Tuna tartare with avocado",

"user_id": 45,

"nutrition": ["nutr_211": "2435.187", "nutr_645": "1968.976"]
},
{
....
```

Figure 3.3: DynamoDB data model

Although relational databases support many data types, DynamoDB attributes support just a few: scalar data types (String, Binary, Number, Boolean, and Null), multivalued data types (string set, binary set and number set) and document (List and Map). A simple data model with some of the described components is shown in fig. 3.3.

Every table's data is stored in *Partition*s that are nothing more than allocation of storage in a SSD.

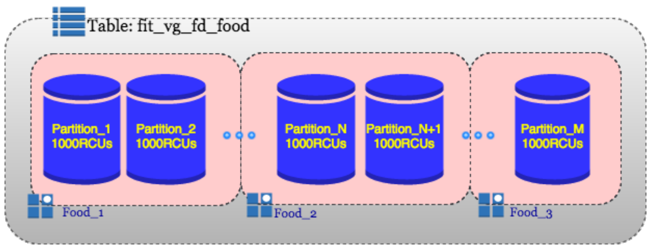


Figure 3.4: Example of table's partitions

The distribution among the partitions is based on the partition key value and the partitioning logic depends upon two things: table size and throughput. To reach the full amount of the specified provisioned throughput, since DynamoDB is *"optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be"* [23], it is strongly recommended to create tables where partition keys have very distinct values in such a way to maintain the workload smoothly spread across those partition key values and distribute the requests across partitions.

Amazon documentation explains that *"if a table has a very small number of heavily accessed partition key values then request traffic is concentrated on a small number of partitions. If the workload is heavily unbalanced, it means that it is disproportionately focused on one or a few partitions and so the requests will not achieve the overall provisioned throughput level"* [23]. Generally, the throughput is more efficiently utilized when the access ratio of partition key values to the total number of partition key values, in the table, increases.

During the creation of a new table the user is required to specify the provisioned throughput capacity that such table needs to be reserved for reads(read capacity units) and writes(write capacity units) [45]:

- One ***Read Capacity Unit*** allows to have one strongly consistent(or two eventually consistent) read/s per second towards items of size up to 4KB. Read capacity unit depends either on item size and whether the user needs strongly or eventually consistency. In case it is required to read larger items, read capacity units will need to be increased(at some price).
- One ***Write Capacity Unit*** depends on the item size since it allows to write items up to 1KB per second. If the item that needs to be written is larger than 1KB than write capacity units need to be increased.

Exception is done for table with Global Secondary Indexes(GSI) that will consume double capacity units, one to write in the table and one to write in the index. It is worth to specify that DynamoDB estimates the number of consumed write/read capacity units based on item size and not on the amount of data that has to be stored or returned to an application. *"The number of capacity units consumed will be the same whether you request all of the attributes (the default behavior) or just some of them using a projection expression"* [20].

Once the table is created DynamoDB automatically allocates a sufficient number of partitions to it in order to be able to handle the specified provisioned throughput(every single partition supports at maximum 3000 read capacity units, 1000 write capacity units and approximately 10GB of data) and from that moment on all the concerns about partition is fully managed by DynamoDB [23]. After the creation, data is also automatically replicated across multiple Availability Zones within an AWS region. In the case that more storage is needed because provisioned throughput exceeds the limit a partition can support or the partition's capacity is fully used, then DynamoDB will allocate additional partitions.

To see how many partitions a new table would require in order to accommodate the provisioned throughput the following equation can be used:

$$\#ThroughputPartitions = \frac{readCapacityUnits}{3000} + \frac{writeCapacityUnits}{1000} \tag{3.1}$$

To calculate the number of partitions to accommodate the table size indeed:

$$\#PartForTableSize = \frac{TableSize(GB)}{10(GB)} \tag{3.2}$$

Eventually the total number of partition to be created is given by the equation:

$$\#Partitions = \mathbf{MAX(}\#ThroughputPartitions | \#PartForTableSize\mathbf{)} \tag{3.3}$$

Example 1:
Supposing we are required to create a new table of about 20GB in size with 600 read capacity units(rcu) and 600 write capacity units(wcu), the initial number of partitions required would be as follow:

$$\#ThroughputPartitions = \frac{600}{3000} + \frac{600}{1000} = 0.8 \rightarrow \mathbf{1}$$

$$\#PartForTableSize = \frac{20}{10} = 2$$

$$\#Partitions = \mathbf{MAX}(1|2) = 2$$

Since it has been created DynamoDB adopts a combination of several well known approaches to obtain such a performing system. It reaches uniform data partitioning through a consistent hashing algorithm and ensures consistency by making use of object versioning and quorum techniques to preserve consistency among the replicas [33]. The DynamoDB system is also properly balanced thanks to its symmetric structure made by a ring of nodes where none of them is master and none of them has extra responsibilities or performs extra work.

3.2 Database manipulation

Once the database and tables are created and populated with the data, every user must be able to manipulate such data and tables as desired. Typical manipulations operations involve Update(Insertion, Deletion, Modification) and Retrieval operations and relational databases as NoSQL have different ways to perform them.

3.2.1 Insert operation

The purpose of an insert operation is to add one or more items to an existing table. An interesting feature is the fact that after an INSERT statement edits are non permanent until a COMMIT statement is issued since SQL databases are transaction-oriented rather in Amazon DynamoDB, every action is permanent once it replies with an HTTP 200 status code.

3.2.1.1 MariaDB

Relational databases present a two-dimensional data structure composed of rows and columns. The relation's name with a list of attributes values have to be provided and listed in the same attributes' order as the one used during the CREATE TABLE command. Only the values of attributes created with the clause NULL or DEFAULT can be omitted. Some RDBMS also support semi-structured data with native JSON or XML data types but it mainly depends on the vendor. As explained in subsection 2.2.1.1 every new item to be inserted does not have to violate integrity constraints specified on the relational schema. Insertion is considered a sensitive operation since every constraints could potentially be violated. It is enough to provide a NULL value for the primary key of the new tuple to violate the Entity integrity. Key constraint could be violated by providing already existing key/s inside the same relation as Referential constraint could be broken by adding a foreign key pointing to a non existing tuple in the referenced table. Domain constraint can also be violated by supplying an attribute's value that does not correspond to the associated domain. SQL language uses **INSERT** statement to add tuples:

```
INSERT INTO fit_vg_fd_food (food_guid, name, user_id,
    nutr_203, nutr_209)
VALUES('625414b7-e77b-4c32-a29d-56d956842c5a',
  'Apple pie prepared from recipe',
  3, 3716.541, 1479.689);
```

If any of the previous constraints is not respected, then the whole operation is rejected.

3.2.1.2 DynamoDB

Write operations in DynamoDB are based on the primary key(s) which is the only attribute strictly required [45]:

- With only a Partition Key, once the request is forwarded, DynamoDB uses the partition key as input to an internal hash function which returns a value representing the partition where the item will be stored.
- With a composite key, the partition where the item will be stored is again calculated with the hash function through the partition key but within the partition the item will be stored in ascending order based by the Sort key among the ones with equal partition key.

Besides the primary keys, there are a couple of other constraints to be respected like attributes values must not be NULL and String/Binary type attributes lengths has to be greater than zero [20]. DynamDB uses the **PutItem** action [45] to insert a new item, or, if the item already exists, to replace the old item with the new one, in an existing table.

```
{
  "TableName": 'fit_vg_fd_food',
    "ConditionExpression": "string",
    "ExpressionAttributeNames": {
      "string" : "string"
    },
    "ExpressionAttributeValues": { ... },
  "Item": {
    "food_guid": {'S': '625414b7-e77b-4c32-a29d-56
    d956842c5a'},
    "name": {'S': 'Apple pie prepared from recipe},
    "user_id": {'N': 3},
    "nutr_203": {'N': 3716.541},
    "nutr_209": {'N': 1479.689},
  },
  "ReturnConsumedCapacity": INDEXES | TOTAL | NONE,
    "ReturnItemCollectionMetrics": "string",
    "ReturnValues": "string"
}
```

Another DynamoDB insertion operation is called **BatchWriteItem** [45] and it allows to insert batches of 25 item(up to 16 MB of data) in parallel in one or more tables.

```
{
   "RequestItems": {
      "fit_vg_fd_food" : [
         {
            "PutRequest": {
               "Item": {
                  "food_guid": {'S': '625414b7-e77b-4
    c32-a29d-56d956842c5a'},
                  "name": {'S': 'Apple pie prepared
    from recipe},
                  "user_id": {'N': 3},
                  "nutr_203": {'N': 3716.541},
                  "nutr_209": {'N': 1479.689},
               }
            }
         }, ... ]
   },
   "ReturnConsumedCapacity": "string",
   "ReturnItemCollectionMetrics": "string"
}
```

If the provisioned throughput will exceed or any other failure breaks the execution, it would return the failed items in a parameter called *UnprocessedItems* so that the request with the unprocessed items can be submitted again. However, due to the throttling on the individual tables, Amazon advices to delay the operation by using exponential backoff algorithm so that *individual requests in the batch are more likely to succeed* [20, 45].

With BatchWriteItem also some constraints have to be respected. First of all, primary key(s) attribute must match the ones from the table schema, then all the tables specified in the request must exist and items cannot exceed the number of 25 in each batch or individual items within a batch cannot exceed 400KB. Moreover, within the same BatchWriteItem operations is not allowed to perform multiple operations(like Insert and Delete) and the total request size must be smaller or equal to 16MB [20, 45].

If any of these constraints is not respected, the whole operation will be rejected.

3.2.2 Update operation

Update operation allows to modify attributes' value(s) related to one or more tuples in a specific table.

3.2.2.1 MariaDB

MariaDB uses the **UPDATE** statement to update items in a DB's table. It requires to specify the relation's name and specific conditions on the attribute of such relation in order to select the ones to be modified. Generally, if no primary/foreign keys are involved, it is enough to check for domain and data type correctness, otherwise, editing a primary key value is equal to delete one tuple and inserting another one, so all the integrity constraints from the INSERT operation have to be validated, especially those who are related to referential constraint [26].

```sql
UPDATE fit_vg_fd_food
SET food_url_id = 'apples raw with skin'
WHERE food_guid = '625414b7-e77b-4c32-a29d-56d956842c5a
    ';
```

3.2.2.2 DynamoDB

DynamoDB uses **UpdateItem** action to edit existing items' attributes' or to add new one in case it is not already present in the specified table [20, 45].

In contrast with relational databases, here attributes can be added or simply updated to meet the business requirements because. Since NoSQL databases are schema-less every item might have not only a different number of attributes but also different attributes' types'(apart from the partition/sort keys).

Following an example of Update request towards the table *fit_vg_fd_food* on the attribute *food_url_id*:

```
{
  "TableName": 'fit_vg_fd_food',
  "Key": {
     'food_guid': {
           'S': '625414b7-e77b-4c32-a29d-56d956842c5a'
     }
  },
     "ConditionExpression": "string",
     "ExpressionAttributeNames": { ... },
  "UpdateExpression": 'set food_url_id = :t',
  "ExpressionAttributeValues": {
    ':t': {'S': 'apples raw with skin'}
  },
  "ReturnValues": 'UPDATED_NEW'
    "ReturnConsumedCapacity": True | False,
    "ReturnItemCollectionMetrics": "string"
}
```

3.2.3 Delete operation

The purpose is to delete an entire record/item from an existing table.

3.2.3.1 MariaDB

MariaDB uses a so called **DELETE** statement to delete existing record in a table. The clause WHERE has to be specified to indicate which record(s) will be deleted, otherwise, the omission of such clause will cause the deletion of all the tuples in the table (**TRUNCATE** operation is designed for this operation indeed). Following an example of a delete operation:

```
DELETE FROM fit_vg_fd_food
WHERE food_guid='625414b7-e77b-4c32-a29d-56d956842c5a';
```

The operation of records' deletion takes time because when a DELETE statement is typed, firstly all the data get copied into the *rollback segment*[1], then the delete get performed. That is the reason why when a "ROLLBACK" is typed after a table has been deleted, the system is still able to get the data back from the Rollback segment. If the table involved in a delete operation is small in size, it is possible to play around and speed things up with the following:

```
LOCK TABLE fit_vg_fd_food WRITE;
DELETE FROM fit_vg_fd_food WHERE food_guid='625414b7-
    e77b-4c32-a29d-56d956842c5a';
UNLOCK TABLES;
```

The LOCK and UNLOCK statements might also be avoided but they helps to prevent potential deadlocks. Unfortunately, this solution may be very slow if the table's size is large because it locks row by row. If a large amount of items have to be deleted the process of locking a row, deleting it, and then locking the next row and also deleting it(and so on) may take longer than a simple delete.

TRUNCATE TABLE always locks the table and page but not each row.) When you issue a delete table it locks a row, deletes it, and then locks the next row and deletes it. Your users are continuing to hit the table as it is happening. I would go with truncate because its almost always faster.

With delete operation only referential integrity may be violated if the record being deleted is referenced by foreign keys from other records in the same DB. If a violation is detected after a delete operation's request, the system can reject the entire operation or attempt to propagate the deletion to the referenced tuples in the other relations. Another option is to edit(set to NULL[only if it is not part of a primary key] or set a reference to another valid tuple) the referencing attribute values which have caused such violation [26].

3.2.3.2 DynamoDB

DynamoDB uses the **DeleteItem** to remove a single existing item by using its primary key and attributes' values of such delete item might be returned in the response under the parameter *ReturnValues* [20].

A conditional delete might also be performed to delete an item with a specific attribute value. If no conditions are specified, the DeleteItem operation is idempotent meaning that running it several times on the same item does not return any error [45].

```
{
  "TableName": 'fit_vg_fd_food',
  "Key": {
    'food_guid': {'S': '625414b7-e77b-4c32-a29d-56
    d956842c5a'},
    'deleted': {'N': 0}
  },
  "ConditionExpression": "string",
    "ExpressionAttributeNames": { ... },
    "ExpressionAttributeValues": { ... },
    "ReturnConsumedCapacity": True | False,
    "ReturnItemCollectionMetrics": "string",
    "ReturnValues": "string",
}
```

Another way to delete items in DynamoDB is by making use of the previously described **BatchWriteItem** in subsection 3.2.1.2. Changing the key from *PutRequest* to *DeleteRequest*, it allows to delete large amount of data(in batch of 25 items per time):

```
{
  "RequestItems": {
    "Table_Name" : [{
               "DeleteRequest": {
                 "Key": {
                     "Primary_Key_Name" :
```

[1]For recovery purposes, the system keeps track of transactional states. The ROLLBACK segment is an object that keeps track of data written in the database, used for *undo* changes when a transaction is rolled back and to avoid that other transactions see uncommitted changes.

```
                        { "Primary_Key_Type": "
      Primary_Key_Value" }
                    }
                },
            }]
    },
    "ReturnConsumedCapacity": INDEXES | TOTAL | NONE,
    "ReturnItemCollectionMetrics": SIZE | NONE
}
```

3.2.4 Read operation

Read operations are used to select a subset of records/items from an existing table. They can be decorated with many clauses to retrieve any kind and any amount of data from one or more tables.

3.2.4.1 MariaDB

MariaDB uses the **SELECT**(expressed by the symbol σ) statement to retrieve rows from a relation(R) under a given logic expressed by the user through one or more selection condition(s) to filter rows of user's requirements. Select operation works almost always in combination with **PROJECT**(expressed by the symbol π) operation which fundamental to specify the required attributes resulting from the query.

These two operations are represented by the following statements:

$$\sigma_{<selection\_conditions>}(R) \tag{3.4}$$

$$\pi_{<attributes_list>}(R) \tag{3.5}$$

The following expression instead presents a basic combination of both SELECT and PROJECT operations:

$$\pi_{<attributes_list>}(\sigma_{<selection\_conditions>}(R)) \tag{3.6}$$

which in turn is translated as:

```
SELECT <attributes_list>
FROM <tables_list>
WHERE <selection_conditions>;
```

Attributes' list could contain any attribute as long as it is present in the specified table(s) and attributes in this list are the ones returned in the response once the query will be executed. Tables' list contains any table name in the database where the attributes specified in attributes_list have to be present. Selection conditions represent the user's choice.

3.2.4.2 DynamoDB

DynamoDB make use of GetItem, BatchGetItem, Query and Scan operations to retrieve items from specified tables matching a provided Primary Key(Partition/ Composite key) or, in case of Query and Scan, matching a Global Secondary Index(explained later in subsection 3.3.2) if the table owns any. The values of the key(s) are used as input to an hash function which returns the partition where the item is stored. If no matching items are found those operation does not return any data in the response [20].

GetItem operation returns attributes of the matching Primary Key from one table. Following an example of GetItem request:

```
{
    "ConsistentRead": True | False,
    "ExpressionAttributeNames": {
        "string" : 'string'
    },
    "Key": {
        "food_guid" : {
            "S": '625414b7-e77b-4c32-a29d-56d956842c5a'}
    },
    "ProjectionExpression": 'string',
    "ReturnConsumedCapacity": TOTAL,
    "TableName": 'fit_vg_fd_food'
}
```

By default GetItem implements eventually consistent read which in turn can be changed to strongly consistent read just by setting the parameter *ConsistenRead* to *True*.

BatchGetItem action returns attributes matching the Primary Key of one or more item(s) in one or more table(s). The size limit of the returned items in the response is set to 16MB and if the limit is exceeded BatchGetItem will return a partial result and the missing items will be returned in the parameter *UnprocessedKeys* to continue retrieving data form the next item(as in subsection 3.2.1.2 is advised to use *exponential backoff algorithms*) [20].

```
{
    "RequestItems": {
        "fit_vg_fd_food" : {
            "AttributesToGet": [ "string" ],
            "ConsistentRead": True | False,
            "ExpressionAttributeNames": {
                "string" : "string"
            },
            "Keys": [
                {
                    "food_guid" : {
                        "S": '625414b7-e77b-4c32-a29d-56
    d956842c5a'
```

```
                }
            }
        ],
        "ProjectionExpression": 'string'
    }
  },
  "ReturnConsumedCapacity": TOTAL
}
```

BatchGetItem will return a *ProvisionedThroughputExceededException* if the Provisioned throughput is not enough to process even one item [20].

Query uses partition key to access data of a table, a local secondary index, or a global secondary index and return a result set. Specific values of the partition key can be specified in the parameter **KeyConditionExpression** to allow the Query operation to return all the matching items. Resulting items are always returned in ascending order by the sort key value(Number of UTF-8 bytes). To filter the results, *FilterExpression* can be applied to determine exactly which items from the resulting ones(after Query has been executed) should be actually returned to the user [20]. If the limit of 1MB on queried items is exceeded, the missing items might be retrieved in subsequent query operations starting from *LastEvaluatedKey* but in this case the result set might be required to be paginated [22]. Following an example of a Query request:

```
{
  "ConsistentRead": True | False,
  "ExclusiveStartKey": {
     "string" : { ... }
  },
  "ExpressionAttributeNames": {
     "string" : "string"
  },
  "KeyConditionExpression": "food_guid = :a and
   deleted = :b",
  "ExpressionAttributeValues": {
     ":a": {'S': '625414b7-e77b-4c32-a29d-56d956842c5a
   '},
     ":b": {'N': 0}
  },
  "FilterExpression": "string",
  "IndexName": "string",
  "Limit": number,
  "ProjectionExpression": "string",
  "ReturnConsumedCapacity": "string",
  "ScanIndexForward": boolean,
  "Select": "string",
  "TableName": 'fit_vg_fd_food'
}
```

Query operation only operates on matching records(not the entire table) with primary key attribute values eventually supporting a collection of comparison operators on key attribute values to refine the look up process. All of this means that the user only pay for the throughput of the items that match, not for everything that is actually scanned [20].

Scan operation allows to query a table or a secondary index returning one or more items and attributes. A big issue with Scan operation is that it firstly goes through the entire table, iterating over every item and filtering out the returned results only afterwards. As with Query operation *FlterExpression* parameter can be set to filter returning items once the complete scan is performed. What is worst is that a single Scan can only retrieve data up to 1 MB, if this limit is exceeded, the remaining results are returned in as *LastEvaluatedKey* to continue to retrieve items in a subsequent Scan operation. Moreover, Scan operation may be really slow as the table(index) size increases

Following an example of Scan request is shown:

```
{
    "TableName": 'fit_vg_fd_food',
    "ConsistentRead": True | False,
    "ExclusiveStartKey": {
        "string" : { ... },
    "ExpressionAttributeNames": {
        "string" : "string"
    },
    "ExpressionAttributeValues": {
        ":a": {'S': '625414b7-e77b-4c32-a29d-56d956842c5a
    '},
        ":b": {'N': 0}
    },
    "FilterExpression": 'food_guid = :a and deleted = :b
    ',
    "IndexName": "string",
    "Limit": number,
    "ProjectionExpression": "string",
    "ReturnConsumedCapacity": "string",
    "Segment": number,
    "Select": "string",
    "TotalSegments": number
}
```

Since by default Scan operation works sequentially scanning the whole table, with large tables(or secondary indexes) it would result to be really slow and high amount of provisioned throughput would be required, so, to reach performance improvements it is advised to set the *TotalSegments* and *Segment* parameters in such a way that parallel Scans operation will be performed.

3.3 Indexing

Indexes represent a supplementary access pattern very useful to speed up the lookup and retrieval of items matching specific search conditions. This section aims to provide a brief presentation of the type of indexes used in MariaDB and in DynamoDB.

3.3.1 MariaDB

Indexes are used to improve lookup times in retrieving specific information more frequently than others. They're usually tree-based, so that looking up a certain row via an index takes O(log(n)) time rather than scanning through the full table.

However, there could be drawbacks if unnecessary indexes are added making INSERT, DELETE and UPDATE operations slower than the usual since the more indexes a table maintains the more extra information for each indexed column MariaDB needs to store. By definition, it is known that any column in a large table that is frequently used in ORDER BY, WHERE and especially JOIN operation should have an index [32].

Imagine a table with no indexes at all. During a lookup towards one or more attribute(s) the DB engine will start from the first row reading sequentially through the whole table to reach the matched rows. The inconvenient here is that the larger is the table, the higher is the cost in time and money. With the presence of an index regarding the desired column(s), object of the lookup, the DB engine might rapidly find the position to explore in the middle of the data file without to browse all the data [32].

However, if the table is small or the queries are performed on big tables that return most or all the rows then indexes are not a plus because sequential reads minimize disk seeks, so, in such cases sequential lookups result faster than working through an index [32].

3.3.2 DynamoDB

Indexes are available also in DynamoDB under the name of Secondary Indexes and with the form of "alternate key" to query data in a table in addition to perform queries towards the primary key. It offers two types of indexes:

- **Local Secondary Indexes**(LSI) allow the user to create up to 5 different sort keys(range attributes) besides the existing Partition Key(that has to remain the same as the one created in the table). They must be created within the creation of the table and once created they cannot be deleted or edited anymore. LSIs support eventual consistency as strong consistency and basically they consume provisioned throughput from the base table in the case of read operation while they consume two write capacity units(one for the table and one for the index) for each of the write operations(update, delete, insert). Last but not the least LSI also allow to "retrieve attributes that are not projected to the index" at the price of higher latency and provisioned throughput costs [21].

- **Global Secondary Indexes**(GSI) represent a new paradigm allowing the user to create up to 5 completely new primary keys(hash key and range key) each one with its own provisioned throughput. GSI might be created whenever the user needs and once they are created they can be edited and deleted at any time. However, they only support eventual consistency and they can only retrieve attributes projected to the index.

3.4 Join operations

JOIN is a fundamental relational algebra operator widely used in relational databases to retrieve data from multiple tables combining matching tuples among the involved relations [26]. As it is well known, most of (if not all) the NoSQL database do not support this operation, not being born as relational database and not having a schema, most of them neither really need it. DynamoDB does not support JOIN operations indeed and since, as other NoSQL databases, is born to deal with the increase of data, with big tables it would be better to avoid JOINs because the latency overhead of an individual key lookup would somewhat large by the reason that the database would need to discover firstly towards which node(s) make the query, then make the query itself and finally wait for the response(in most of the cases, this process would create a huge amount of foreign keys lookups through many different nodes). Moreover, if the application needs to perform large amount of JOIN operation may be worth to denormalize[2] the data. It is a common use in NoSQL databases to have redundant data in different places to make lookups easier and faster. Moreover, most of non relational databases do not support secondary indexes either meaning that duplicated stuff is strictly required if the application needs to make queries by other criterion.

DynamoDB should maintain denormalized data, same keys could even be stored across multiple tables but NoSQL DBs will not synchronize them automatically and they will not have any foreign-key meaning, even if they exists, they will be technically considered as a different set than ones in other tables and it would only be a task of the application itself to perform such synchronization. Having said that, in most of the cases for a NoSQL database like DynamoDB having normalized or denormalized data depends on the application use cases.

On the one hand, data could be normalized if:

- every item stored exceed the limit of 400KB and every item could be stored in different tables or in Amazon **S3**[3].
- the application needs to perform a lot of writes/updates and persistent writes of/on item(s) larger than 1KB will impact the amount of write capacity units(that have 1KB limit per item size) even if only one attribute is edited the WCUs considered will be the one of the entire item.

On the other hand, data's normalization could be useful if:

[2]Denormaliation is a strategy used to improve lookups in a database by having redundant(or grouped) data and loosing something in write performance [26].

[3]`https://aws.amazon.com/documentation/s3/`

- items to be stored have small size and it should maintain just a few attributes. For reads operation an item should be equal or smaller than 4KB, to not exceed the fixed limit of read capacity units while for write the limit is 1KB.
- the application has high-traffic and consistency as data synchronization among tables is not its main concern.

3.5 Highlights

Besides the issue about Joins that are of course impossible and so complex data relations have to be managed on the code/cache layer, the first thing to come to an hand is the fact that querying data in DynamoDB is extremely limited especially in the need to query non-indexed data. Indeed queries can be only performed towards a single hash, on a hash/range key, or an index and only up to 5 secondary global indexes can be created. Moreover, when it comes to anything beyond the most basic key/value queries, DynamoDB looks still a bit crippled. While in MariaDB, besides being possible to make queries towards any attributes in the table, there are no specific limits on the number of indexes(single or composite) to be created. Mainly indexes in DynamoDB are not built to speed up tables' lookups rather to query the table towards attributes different from the original primary key specified at the table creation step. Another aspect which comes up from the data models is the difference in number of supported data types between the two databases. For sure it will be necessary to parse some attributes values to build unfailing queries.

All these aspects have to be taken into account during the framework design where all the system is set up, the tables are created and the queries are built to perform the benchmarking analysis.

Chapter 4

Benchmarking Framework Implementation

To perform our benchmarking analysis towards Amazon RDS and Amazon DynamoDB, a full scale testing on both online versions where required. Amazon also provides a local version of DynamoDB but it mainly has been developed as a sort of ”getting to know” to help developers in getting familiarity with the Amazon NoSQL DB. Moreover, getting in contact with DynamoDB’s developers, also them have advised not to perform analysis on local version since the results may fluctuate drastically.

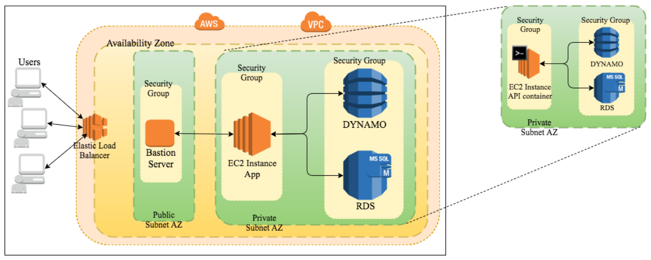


Figure 4.1: Example of a simple AWS application structure

AWS applications, in general, involve a lot of different components interacting among them, setting up connections, exchanging messages and informations. To avoid any traffic interaction time in our analysis the main idea was to log all the benchmarking analysis at database level so that the real query time(and only that) could be monitored and recorded. Although this could have been easily done with RDS, DynamoDB does not support any kind of database level logging and due to this, the plan has been deviated in favour of the API level. In order to do that, and to deal with some issues such as DB caching, TCP times, connection times, authentication (and so on) a proper framework has been implemented.

Fig. 4.1 shows a really simplified AWS application structure and the extracted components used in this work. The API is maintained in the EC2 instance and it is directly connected to both Amazon RDS instance and Amazon DynamoDB. All of this happens within a private subnet availability zone.

As it has already been explained in the previous chapter, since most of the useful operations from relational databases, such as JOIN, AVG, MIN, MAX are not supported by DynamoDB (and NoSQL DBs in general), the benchmarking analysis framework on this work has been built essentially on five basic operations to be performed, that are Read, Write(Insert and Update), Delete and Query on Indexes.

All the queries shown in subsections 4.2.2 and 4.2.3 have been performed across all the tables involved in both databases retrieving, updating and deleting items through their Primary Key/Indexes or inserting by providing a **Food object** built through the **Factories**[5]. The time recorded from every API calls is the time that starts when the query is issued until a valid response is returned back by the database.

4.1 Caching

Amazon RDS makes use of database caching which allows to store results of SELECT queries so that if an identical query is received in future, the results can be quickly returned. In order to avoid that, for a fair comparison, caching had to be disabled in both databases. Connecting to the RDS instance it is pretty simple to check whether the query cache is available and enabled, by just running the following query:

```
SHOW VARIABLES LIKE 'have_query_cache';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+
```

If the query returns *YES* as in the result above, then the query cache is available. To disabled it, it is enough just to set a couple of variables to 0. This has been done by running:

```
set global query_cache_type=0;
set global query_cache_size=0;
flush query cache;
reset query cache;
```

From now on, the RDS cache is disabled so every query can be run fairly under this point of view. From the AWS documentation and AWS support, DynamoDB does not provide any caching service.

[5]http://factoryboy.readthedocs.io/en/latest/orms.html

4.2 API level(Django Framework)

Django framework has been chosen as high-level Python Web framework since it is free of charge, open source, it is provided by a really complete documentation and it takes care of many concerns about Web development. This framework also allows the developers to create custom fully integrated commands to be executed from the instance's terminal and for the purpose of this work this feature was perfect since it was possible to create a command to be run for each of the operation involved in the benchmarking analysis.

Django provides an official backend[1] for relational databases to take care about setup and connections while it provides models to allow any interaction between the application and the database instance. However, Django does not provide any implemented backend for NoSQL DBs and unfortunately to build an entire backend from the scratch would have meant to rewrite the whole Django ORM which takes a huge amount of work, time and effort.

Playing around the inconvenient, luckily Amazon released an interesting SDK for Python called **boto3** which allows to get the connection to DynamoDB and to perform operations on it. "Boto is the Amazon Web Services (AWS) SDK for Python that provides an easy to use, object-oriented API as well as low-level direct service access" [19].

4.2.1 Operation's definitions

All the operations taken into account in this work are defined as follow:

⋆ **Reads.**

- Read the value corresponding to a given key. This is the same as the read operation in the CRUD model.
- Read the value/values corresponding to a non-key(index or general attribute)

⋆ **Write.**

- Save a new item in the table. This represents the same as the insert operation in the CRUD model.
- If a given key-value pair is not found in the table, then the item is added to the table. Otherwise, it updates the value for the given key. This operation combines Create and Update operations of the CRUD model.

⋆ **Update.**

[1]Django provides some basic well-documented implementations of authentication, caching, database setup, setting variables (and so on) to anticipate the need of adding own custom "backend" implementation and own custom modules.

  - Update an item in the table. This is the same as the update operation in the CRUD model.
  - If a given key-value pair is found, then it updates the provided value for the given key.

- ⋆ **Delete.**
  - Delete the record corresponding to given primary key. This is the same as the delete operation in the CRUD model.
  - If the key is not found, then an empty response is returned.

- ⋆ **Read by Index.** Retrieve one or more item(s) corresponding to a given index value. This could be thought as the same as the read operation in the CRUD model but by providing an index instead of a primary key.

- ⋆ **Read by Index with Conditions.** Retrieve one or more item(s) corresponding to the given index value and matching a specified condition.

Besides these queries to evaluate scalability and performance on items' requests also throttling, errors and Global Secondary Index creation have been treated and monitored.

The query time of each operation was simply recorded through a straightforward Python module called **time**[2] [38]. The following lines of code show the actual calculation of the query time used throughout the benchmarking analysis:

```
listQueryTimes = []
t = time.time()
"""QUERY TO BE PERFORMED"""
newTime = (time.time() - t) * 1000
listQueryTimes.append(newTime)
```

Although the time is returned as floating point value, not all the system can assure better precision of 1 second so, as workaround of possible issues, the time has been always recorded in milliseconds by simply multiplying per 1000 the difference between the start and stop values returned from *time* function.

4.2.2 Django models

In a Django web application, data is accessed and managed through Python objects known under the name of **models**. Django models map to database tables defining the structure of the stored data, including field types, sizes, default values, etc. and providing an "environment" to encapsulate business logic. This helps developers to have a more structured logic without having to deal with long complex queries built in plain SQL. Once a database engine is chosen

[2]**time.time()** *returns the time in seconds since the epoch as a floating point number. Even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.*

and settled, Django takes care about all the communications and transactions and developers can concentrate their efforts in building the application making all the queries through the database-abstraction API provided by Django.

For the purpose of our analysis a model to map the table *fit_vg_fd_food*(object of this work) has been created as **class FoodBase(models.Model)**. The name base has been given because the model is actually a base to create all the other needed tables in such a way they are all equals but with different sizes.

Once models are created, to let Django the use of them, it is enough to set the **INSTALLED_APPS** parameter by adding the name of the module which contains the created models.

Django uses the class **QuerySet**[3] to create queries towards its models:

```
class QuerySet(model=None, query=None, using=None)[
    source]
```

Most of these QuerySets return other QuerySet objects allowing to chain refinements together in order to perform a sort of complex nested queries.

The operations defined in subsection 4.2.1, in Django are performed as follows:

4.2.2.1 Write

Both the following insert operations come from the description in subsection 3.2.1.1:

- **Insert a single item**. Django uses the class **Model(**kwargs)** [36] to build the object and the *keyword arguments* are simply the names of the fields(attributes) defined in the **Food** model.

  ```
  Food(**food).save()
  ```

  In this case the keyword argument ***food* represents a dictionary containing keys and values of the item that has to be saved in the table. The method **save()** eventually performs the operation to insert the built object in the table.

- **Insert batch of items** Objects are saved in batch through the QuerySet **bulk_create(objs, batch_size=None)** [36]. The list *objs* may contain "any" number of items to be inserted in the table and in the case of this work is expressed by *rdsBulkList*:

  ```
  Food.objects.bulk_create( rdsBulkList[index:index + 25] )
  ```

  To have a fair comparison has been decided to limit the batch at 25 elements since DynamoDB has this limit with *batch_write_item* operation.

[3] https://docs.djangoproject.com/en/1.11/ref/models/querysets/

4.2.2.2 Update

Firstly the item(s) to be updated are filtered by primary key through the QuerySet **filter(**kwargs)** and it returns the QuerySet containing objects matching the given lookup parameters [36].

```
Food.objects.filter( guid=guid )
             .update( food_url_id='apples-raw-with-skin' )
```

As in subsection 3.2.2.1 objects are updated through the QuerySet **update(**kwargs)** [36] where *kwargs* contains the attribute(s) to be updated.

4.2.2.3 Delete

As in subsection 3.2.3.1, objects are deleted through the QuerySet **delete()** [36]. The item(s) to be deleted are firstly selected through the QuerySet **filter**.

```
Food.objects.filter( guid=keys[i], deleted=0 ).delete()
```

With *Foreign keys* Django adopts ON DELETE CASCADE feature deleting also any object pointing at the objects to be deleted.

4.2.2.4 Read

As in subsection 3.2.4.1, objects are retrieved from the database through the QuerySet **get(**kwargs)** [36]. The following primary key

```
key = {
  'food_guid': { 'S': '871af1c4-1e32-4278-91ab-0a31ae83abc8' },
  'deleted': { 'N': 0 }
}
```

is given. In this query the *keyword arguments* are represented by only the composite primary key (*guid*, *deleted)*:

```
Food.objects.get( guid=key['food_guid']['S'], deleted=0 )
```

4.2.2.5 Operation on indexes

- **Query on index**. Objects are retrieved from the database with the QuerySet **get(**kwargs)** where the *keyword arguments* are represented by a single index attribute:

  ```
  Food.objects.filter( user = 3 )
  ```

- **Query on index_with_conditions**. Objects are retrieved from the database through the Query Related tool **class Q**[**source**]. Q() objects allow the creation of complex query with conditions:

  ```
  Food.objects.filter( Q(user=3) &
                       Q(food_type__lte=3, food_type__gte=1) &
                       Q(deleted=0) )
  ```

The previous query is in charge to retrieve all the food items matching the following combinations:

```
user_id = 3, 1 < food_type < 3 and deleted = 0
```

4.2.3 Boto 3 SDK for DynamoDB

While, in Django framework, provided models have been used, in DynamoDB a wrapper class called **DynamoDBManager** has been created to deal with the connection and to collect all the operations required to perform our benchmarking analysis properly allowing to easily manage all the boto3 calls in every script. Boto 3 provides a *resource* or a low level low-level *client* to represent Amazon DynamoDB and operates on the tables. For this work client has been chosen to have better performance benchmarking since resource keeps trying to query if the operation does not succeed straight away while client returns an error.

```
class DynamoDBManager:
  def __init__(self, table_name):
      self.dynamo=boto3.client('dynamodb', region_name='eu-west-1')
      self.table=table_name

  def insert(self, item): ...

  def insert_batch(self, item): ...

  def delete_item_by_key(self, key): ...

  def delete_batch(self, keys): ...

  def update(self, key, update_expression, expression_values): ...

  def get_item(self, key): ...

  def batch_get_item(self, key): ...

  def query(self, key_condition, expression_value): ...

  def query_by_index(self, index, key_condition, expression_value):
      ...

  def query_index_condition(self, index, key_condition,
    expression_values, filter_expression): ...

  def scan(self, filter_expression, expression_value): ...
```

The *DynamoDBManager* class makes use of the class **DynamoDB.Client** as a low-level client representing Amazon DynamoDB and a table name. To create an object with *DynamoDBManager* an existing table name has to be provided. Once the object is created it can make use of all the methods provided by the

DynamoDB.Client(some of them, the ones used in this work, are provided in the class as methods of the object itself)[4]. Following all the operations used in this work are presented.

4.2.3.1 Write

Both the following insert operations come from the description in Subsection 3.2.1.2:

- Insert a single element with the method **put_item(**kwargs)**. Keyword arguments are represented by the table name and the item to be inserted in the form exposed in fig. 4.3.

```
response = self.dynamo.put_item(
  TableName=self.table,
  Item=item
)
```

- Insert a batch of 25 items with the method **batch_write_item(**kwargs)**. Keyword arguments are represented by the table name and a list of items each one in a dictionary shape as shown in fig. 4.3.

```
response = self.dynamo.batch_write_item(
  RequestItems={
    self.table: items
  }
)
```

4.2.3.2 Update

The update operation makes use of the method **update_item(**kwargs)** largely discussed in Subsection 4.2.3.2(3.2.2)

```
response = client.update_item(
  TableName=self.table,
  Key={
    'food_guid': {'S': str(guid)},
    'deleted': {'N': 0}
  },
  UpdateExpression="set food_url_id = :t",
  ExpressionAttributeValues={
    ':t': {'S': str('apples-raw-with-skin')}
  },
  ReturnValues="UPDATED_NEW"
)
```

The *keyword arguments* passed to the method is made by five parameters: the table name, the primary key needed to specify the item to be updated, a statement containing the attribute name to be updated, the new value to be set in the specified attribute and a parameter *ReturnValues* that has been set to UPDATED_NEW to return the updated items in the response(useful to check if the query has been executed properly).

[4]http://boto3.readthedocs.io/en/latest/reference/services/dynamodb.html

4.2.3.3 Delete

Both the following delete operations come from the description in subsection 3.2.3.2

- **delete_item(**kwargs)** method is used by DynamoDB to perform the deletion operation of a single item. It takes as keyword arguments the table name and the primary key of the item to be deleted.

```
response = self.dynamo.delete_item(
  TableName=self.table,
  Key={ 'food_guid': { 'S': str(guid) },
        'deleted': { 'N': 0 } }
)
```

- **batch_write_item(**kwargs)** is method used by DynamoDB to delete batch of items. It takes as keyword arguments the name of the table and a list of primary keys to be deleted from such table.

```
items = []
for i in range(len(keys)):
  items.append({
    'DeleteRequest': {
      'Key': { 'food_guid': { 'S': keys[i] },
               'deleted': { 'N': 0 } }
    }
  })
  t = time.time()
  response = self.dynamo.batch_write_item(
    RequestItems={ self.table: items }
)
```

4.2.3.4 Read

As presented in subsection 3.2.4.2 DynamoDB owns four methods to retrieve items from the tables. The following primary key was provided:

```
key = {
  'food_guid': { 'S': '871af1c4-1e32-4278-91ab-0a31ae83abc8' },
  'deleted': { 'N': 0 }
}
```

- DynamoDB retrieves item(s) corresponding to a given primary key through the method **get_item(**kwargs)**, which takes as arguments such primary key and the table name.

```
response = self.dynamo.get_item(
  TableName=self.table,
  Key=key,
  ConsistentRead=True
)
```

  The parameter *ConsistentRead* has been set to True to have strongly consistent reads and so to read from an updated table.

- To return attributes from one or more items in one or more tables DynamoDB uses the method **batch_get_item(**kwargs)**. Same line of *get_item()* but with a list of keys passed as parameter.

```
response = self.dynamo.batch_get_item(
  RequestItems={
    self.table: { 'Keys': [key] }
  }
)
```

- To access items of a the table through either a primary key or index DynamoDB uses the method **query(**kwargs)**. Keyword arguments are the table name, the key conditions and the required values to compute the condition expression.

```
response = response = self.dynamo.query(
  TableName=self.table,
  KeyConditionExpression='food_guid = :a AND deleted = :b',
  ExpressionAttributeValues={ ":a": key['food_guid'],
                              ":b": key['deleted'] }
)
```

- **scan(**kwargs)** represent a really expensive operation that could also be used to retrieve items from a table but it first accesses every item in such table(or secondary index). As *query* operations, the keyword arguments are the table name, the key conditions and values for the condition expression.

```
response = self.dynamo.scan(
  TableName=self.table,
  FilterExpression='food_guid = :a and deleted = :b',
  ExpressionAttributeValues={ ":a": key['food_guid'],
                              ":b": key['deleted'] }
)
```

4.2.3.5 Operation on indexes

To query items by index(GSI different attribute from the primary key) has been decided to use ***query(**kwargs)*** because *scan* operation required too high read capacity units and it anyway took a really high query time.

- **By only index**.

```
response = self.dynamo.query(
  TableName=self.table,
  IndexName='user_id-index',
  KeyConditionExpression='user_id = :u',
  ExpressionAttributeValues={ ":u": {'N': str(3)} }
)
```

  The *keyword args* are the table name, the index name, the key condition represented by the index and the value for the condition expression.

- **By index with conditions**.

```
response = self.dynamo.query(
  TableName=self.table,
  IndexName='user_id-index',
  KeyConditionExpression='user_id = :u',
  ExpressionAttributeValues={ ':u': { 'N': str(3) },
                              ':g': { 'N': str(1) },
                              ':l': { 'N': str(3) },
                              ':d': { 'N': str(0) }
                              },
  FilterExpression='food_type BETWEEN :g and :l AND deleted=:d
    '
)
```

The *keyword args* are the table name, the index name, the key condition expression, a dictionary for the expression attributes values and the filter expression for such values.

4.2.4 Database Tables

In each database have been created four tables of different sizes, respectively 100 thousands, 1 million, 10 millions and 75 million items. For the purpose of these analysis has been decided to populate all of the tables with the same data migrated in parallel from a real table, *fit_vg_fd_food*, containing real data from the cloud. To migrate data in parallel has been used a multiprocessing environment with 8 CPUs and 32GB of RAM. Every process was in charge to migrate around 50 thousands of items(each one of about $1KB$ in size) at time not to bubble over the DyanamoDB write capacity units that have been set to 5000 in all the tables to speed up the process. Further description and details of the migration process are out of the purpose of this work. The following table shows the final setup of the tables with their final size and number of items:

Table Name	Table Size	Number of items
fit_vg_fd_food	71.58 GB	∼ 75 millions
food_100_ths	111.90 MB	∼ 115 thousands
food_1_mln	966.64 MB	∼ 1 million
food_10_mln	9.55 GB	∼ 10 millions

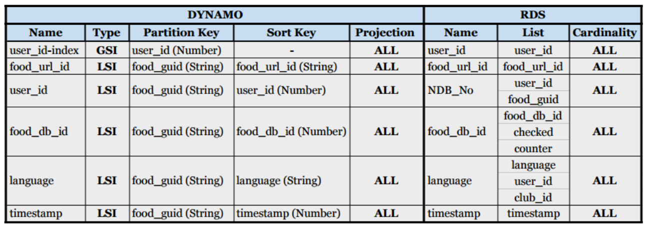
The tables used in Amazon MariaDB roughly reflects the one used in production with same indexes, data and settings. Tables used in Amazon DynamoDB needed to be adjusted in matters of indexes and data types presented in subsections 4.2.5 and 4.2.6.

The Primary Key in all the tables have been thought to be a composite key made by:

- a string value **food_guid**(e.g. 0cc14119-c723-4687-9d76-80ddc4438a76)
- an integer value **deleted**(e.g. 0 or 1)

4.2.5 Indexes

To reflect the real table, several indexes had to be also added:



DYNAMO					RDS		
Name	**Type**	**Partition Key**	**Sort Key**	**Projection**	**Name**	**List**	**Cardinality**
user_id-index	**GSI**	user_id (Number)	-	**ALL**	user_id	user_id	**ALL**
food_url_id	**LSI**	food_guid (String)	food_url_id (String)	**ALL**	food_url_id	food_url_id	**ALL**
user_id	**LSI**	food_guid (String)	user_id (Number)	**ALL**	NDB_No	user_id food_guid	**ALL**
food_db_id	**LSI**	food_guid (String)	food_db_id (Number)	**ALL**	food_db_id	food_db_id checked counter	**ALL**
language	**LSI**	food_guid (String)	language (String)	**ALL**	language	language user_id club_id	**ALL**
timestamp	**LSI**	food_guid (String)	timestamp (Number)	**ALL**	timestamp	timestamp	**ALL**

Figure 4.2: Indexes created in the tables

As described in subsection 3.3.2 DynamoDB allows the creation of only five Global Secondary Indexes and other five Local Secondary Indexes while MariaDB has no fixed limits. Also the meaning of these indexes is a bit different, DynamoDB indexes are mainly thought to allows lookups through different primary keys and not to speed up the lookups as in MariaDB.

From Amazon Best Practices for Local Secondary Indexes[1] there are some precautions to take into account in matter of indexes. Adding or updating an item in a table which contains also LSI will also affect the update/creation of any LSI involved making the indexes growing fastly. A good advice is not to create and maintain multiple indexes if the table is frequently updated and to keep the size of the index as small as possible projecting it only in the most returned attributes because unused indexes contribute to increase storage and I/O costs. Unfortunately, the 95% of the attributes from the table are most of the time needed in the real app to populate the UI so the projection to ALL in the chosen indexes was required. In table 4.2 the chosen indexes are shown.

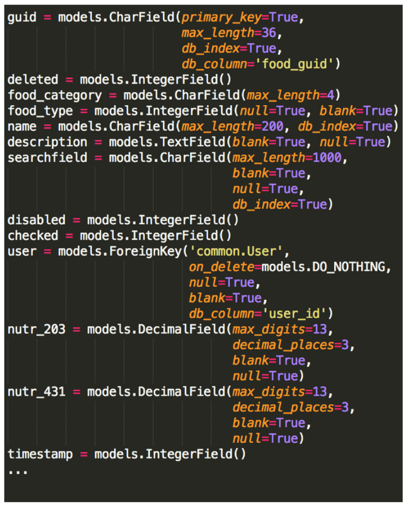
Since queries in DynamoDB can be made only towards primary key or indexes, if it is required to query a different attribute is not possible unless such attribute is part of an index. Besides LSIs that mainly allow to create a new primary key by only choosing a different Sort Key but still the Hash Key has to be the same, Global Secondary Indexes(in subsection 3.3.2) are the components that may do the trick. The attribute to be required can be created as GSI and so being queried directly without any Hash Key need. For the purpose of this work a GSI index was enough so the most used one(*user_id*) has been chosen for the job.

4.2.6 Items' types

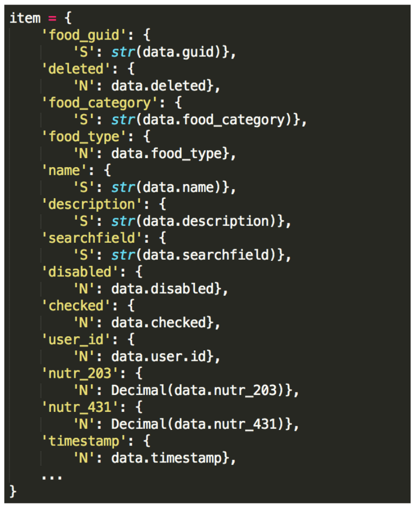
As partially shown in fig. 4.3 each record is made by 57 attributes. A concern to take into account while working with DynamoDB is about data types. Re-

[1]`http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GuidelinesForLSI.html`

lational databases support a huge amount of data types while NoSQLs (in this case DynamoDB) do not.



```python
guid = models.CharField(primary_key=True,
                        max_length=36,
                        db_index=True,
                        db_column='food_guid')
deleted = models.IntegerField()
food_category = models.CharField(max_length=4)
food_type = models.IntegerField(null=True, blank=True)
name = models.CharField(max_length=200, db_index=True)
description = models.TextField(blank=True, null=True)
searchfield = models.CharField(max_length=1000,
                               blank=True,
                               null=True,
                               db_index=True)
disabled = models.IntegerField()
checked = models.IntegerField()
user = models.ForeignKey('common.User',
                         on_delete=models.DO_NOTHING,
                         null=True,
                         blank=True,
                         db_column='user_id')
nutr_203 = models.DecimalField(max_digits=13,
                               decimal_places=3,
                               blank=True,
                               null=True)
nutr_431 = models.DecimalField(max_digits=13,
                               decimal_places=3,
                               blank=True,
                               null=True)
timestamp = models.IntegerField()
...
```



```python
item = {
    'food_guid': {
        'S': str(data.guid)},
    'deleted': {
        'N': data.deleted},
    'food_category': {
        'S': str(data.food_category)},
    'food_type': {
        'N': data.food_type},
    'name': {
        'S': str(data.name)},
    'description': {
        'S': str(data.description)},
    'searchfield': {
        'S': str(data.searchfield)},
    'disabled': {
        'N': data.disabled},
    'checked': {
        'N': data.checked},
    'user_id': {
        'N': data.user.id},
    'nutr_203': {
        'N': Decimal(data.nutr_203)},
    'nutr_431': {
        'N': Decimal(data.nutr_431)},
    'timestamp': {
        'N': data.timestamp},
    ...
}
```

Figure 4.3
Example of Django model(sx) and item from DynamoDB(dx)

Data types supported by DynamoDB are just a few compared with MariaDB:

- **Scalar**: Number, String, Binary, Boolean, and Null
- **Multi-valued**: String Set, Number Set, and Binary Set
- **Document**: List and Map

Even if less types are supported, these seem to be enough to collect all the required data. Sometimes workaround can be found from the API level, as for DATETIME that can be easily converted in a timestamp as integer to be saved in the database and then it could be converted back in plain date after a response back from the db when it is requested from the API.

4.3 DB Connection time, logic behind the schema

As it has already been said in the intro of this chapter, benchmarking analysis performed from the API level are affected by several components involved in the connection process towards the DB such as the three-way handshake, authorization, authentication, etc.

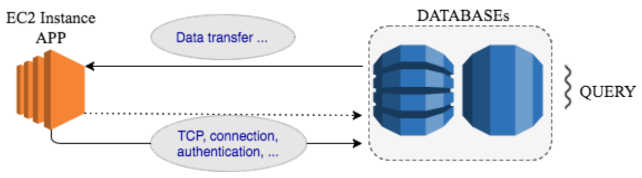


Figure 4.4: APP - DBs transactions

Being conscious to what extent these values can interfere with the actual query time it is fundamental. To measure the connection time it has been decided to run a million times a very simple query(select) towards a table containing only one element. Doing so, the time it takes to be executed from the database engine is negligible and most of the time is taken by the connection process.

Throughout the project the actual time is simply taken by using a python module called *time*. It is called just before the query call to get the starting timestamp λ_1 and right after the same query to get the stopping timestamp λ_2. Having those timestamps, the amount of time between the two is calculated with a simple subtraction getting δ(Eq. 4.1). Summing n times δ it becomes possible to calculate its average that goes under the name of Θ(Eq. 4.2). Furthermore, to have a more refined estimation of the connection time, Θ has been computed several times(m) and its average has been also calculated ending up with an overall average called Γ(Eq. 4.3). (Details of the equation below can be found in table 4.1)

Variable	Description	Value
n	N. of calls to the database	2000
m	N. of cycles	5
λ_1	Start time counter	time.time(ms)
λ_2	Stop time counter	time.time(ms)
δ	Range of time between start counter and stop	$\lambda_2 - \lambda_1$
Θ	Sum of the timestamps from the n calls	$-$
Γ	Sum of the Θ values calculated in the m cycles	$-$

Table 4.1: Formulas' legenda.

$$\delta = \lambda_2 - \lambda_1 \tag{4.1}$$

$$\Theta = \frac{1}{n}\sum_{i=1}^{n} \delta_i \tag{4.2}$$

$$\Gamma = \frac{1}{m}\sum_{i=1}^{m} \Theta_i \tag{4.3}$$

4.3.1 Used queries

The only table used in both RDS and DynamoDB during the connection analysis is(*fit_vg_fd_food*). Such table stores only one simple record.

In RDS the simplest operation to retrieve one item by its primary key is **get**(in code 4.2.2.4).

In DynamoDB, Boto3 provides a function to get only the connection(in code 4.4) without involving any table. However, to maintain the same logic just like the one used for RDS also the query time to retrieve one item has been estimated in DynamoDB. Since Boto3 provides four methods described in subsection 3.2.4.2 to fetch an item from a specified table the connection times were provisioned with all of them: **get_item()**, **batch_get_item()**, **query()**, **scan()**.

4.3.2 TCP ping

Another interesting factor is the TCP ping from the EC2 instance to the DBs to test the reachability of the service on a host using TCP/IP and measure the time it takes to connect to the specified port.

Unfortunately, this test can only be performed towards DynamoDB because RDS instances are not configured to accept(and respond) to an ICMP packet for pings. The only way to establish connectivity to an RDS instance is through a standard SQL client application.

In DynamoDB it is possible to compute the TCP ping by using a Python module called **socket** [61] that is ideal for low-level networking interface and provides an high-level function called *create_connection* [37] which helps to set the connection specifying the host to be pinged, the port and the optional timeout parameter:

```
socket.create_connection(
    ('dynamodb.eu-west-1.amazonaws.com', 80), 1
)
```

Following is shown the method used in the framework to calculate the TCP ping in our benchmarking analysis.

```
def tcp_ping(host, n):
    total_ms = 0
    for i in range(0, n):
        try:
            t_open = time.time()
            s = socket.create_connection((host, 80), 1)  # aws
            s.close()
        finally:
            t_close = time.time()
        ping_ms = (t_close - t_open) * 1000
        total_ms += ping_ms
    return (total_ms / n)
```

By using the equations explained above (4.1, 4.2, 4.3) the estimated time needed it is pretty easy to compute.

4.4 Benchmarking operators

Round-trip times of requests from API level can ”bounce” up and down. This is the reason why every query has been run for many times and every timestamp has been recorded and saved in a list. Once the list has been created three methods have been used. To get a good approximation of the query time the average of the timestamp’s list has been calculated and along with this also the standard deviation to have a precise measure of the mathematical dispersion of the query times from the mean.

- **Mean**. To calculate the average a formula from the original algorithm to compute the weighted average of RTT in the **Adaptive Retransmission Based on Round-Trip Time** [48] has been applied:

  $$OldTimestamp = (\alpha * OldTimestamp) + ((1-\alpha) * NewTimestamp) \quad (4.4)$$

  In the formula, α represents a smoothing factor with a value between 0 and 1. Values of α closer to 1 provide a better smooth in situation where the query time fluctuate wildly. In contrast values of α closer to 0 make the query time change more quickly in reaction to changes in measured query times. Between 0.8 and 0.9 (typically 0.875) is the ”optimal” value.

```
def mean(timestampList):
    alpha = 0.875
    n = len(timestampList)
    OldTimestamp = timestampList[0]
    for i in range(1, n):
        OldTimestamp = (alpha * OldTimestamp) + ((1 - alpha) *
         timestampList[i])
    return OldTime
```

  The function **mean** accepts a list of timestamps taken from the performed queries as argument. Then it applies the formula in Eq.4.4 to calculate the mean throughout the provided list.

- **Variance**.

  $$\sigma^2 = \frac{\sum_{n=1}^{n}(X-\mu)^2}{n} \quad (4.5)$$

  where μ is the mean of the query times(from Eq.4.4) and n is the total number of query times recorded in each query test.

```
def variance(data, mu):
    n = len(data)
    if n < 2:
        raise ValueError('variance requires at least two data
    points')
    v = (sum((x - mu)**2 for x in data))/n
    return v
```

- **Standard Deviation**. Once the mean and the variance are calculated, it is pretty easy to compute the standard deviation(σ) that is represented by the square root of the variance σ^2:

$$\sigma = \sqrt[2]{\sigma^2} \tag{4.6}$$

The Standard deviation helps to understand how good are the found means in respect to the amount of query times recorded.

```python
def std_dev(data, mu):
    var = variance(data, mu)
    return math.sqrt(var)
```

- **DynamoDB connection**. With this operation it is possible to have a rough estimation of the setup time required to build the connection to Amazon DynamoDB from API level created through the client. It could be interesting to see if this plain connection is in line with the connection times examined in subsection 4.3.

```python
def dynamo_connection(db, region):
    t_send = time.time()
    boto3.client(db, region_name=region)
    return (time.time() - t_send) * 1000
```

Chapter 5

Discussion of analysis

Unfortunately, there are no universal metrics to measure performance of key-values solutions due to their different goals and capabilities. It is clear that trying to evaluate a database as DynamoDB, which is designed for write availability above everything else, in the area of consistency, is not fair since the overall system has been create to enforce availability above consistency.

Beside this, one of the most important concerns about web application is the time required to complete a task. In case of a database, this factor is redirected to the time required to complete a transaction. Analysing this time along with scalability and throughput it turned out that all these components may help to get some requirements and eventually to provide an accurate overview and deep understanding of the examined solutions.

The main aim of this work is to present a benchmarking analysis from the requirement of the company **Virtuagym** that is trying to solve the problem of slow queries in some transactions due to the day by day increase in size of some specific tables.

5.1 Connection measurements

From fig.5.1, the connection time to DynamoDB from the EC2 instance reach a peak at about $33.8ms$ in the first connection then it tends to balance to about $1.9ms$ in all the other connection tests. This happen because when a command towards DynamoDB is performed, it firstly needs to verify the AWS credential and then to make resources available for the tables to read while for every subsequent call it will reuse those resources. Indeed, running the script many times(by calling it from the Django command *manage.py*) it always takes around $30ms$(which is roughly the same amount of time compare with the peak), but when the call is made several times in a for loop within the same script, each call's time decrease drastically until a period of no operation where Dynamo seems to close down the resources to save on it. For this reason and for the purpose of our benchmarks analysis the very first call to DynamoDB should always be discarded.

The TCP ping time to DynamoDB is almost always around $0.85ms$ in average, meaning that DynamoDB instance is fastly reachable.

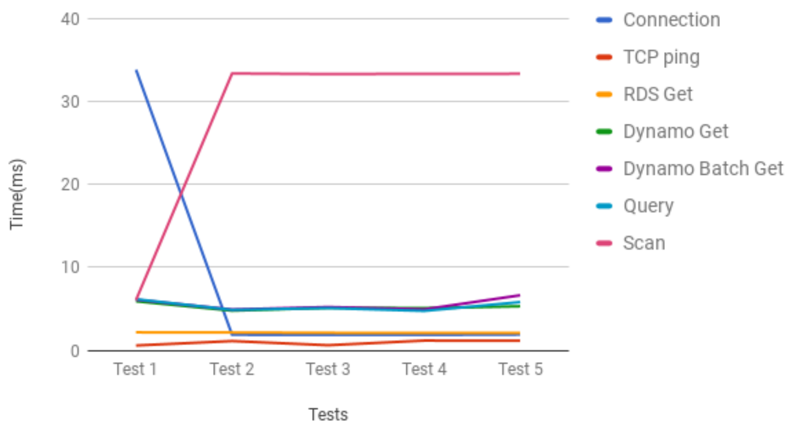


Figure 5.1
Connections times(table B.1)

Query time to get an item from an RDS instance seems to be pretty balanced along all the calls and cycles keeping on track at about $2.16ms$ while all the query times to DynamoDB fluctuate a lot due to the many interactions(connections, latencies, packet exchanges) happening in the backend side of DynamoDB. However, since DynamoDB has several methods to retrieve a record from a table, it was interesting to check if the time every method takes to get the same item was roughly the same or extremely different. As shown in fig. 5.1, most of the times are pretty close in time, between $2ms$ and $7ms$, apart from *Scan* operation that in average takes around $28ms$.

5.2 Key metrics

The following table shows some of the key metrics considered throughout the benchmarking analysis:

Name	Description
Successful Request Latency	Response time (in ms) of successful requests in the selected time period to metric the performance. Also can report an estimated number of successful requests (data samples).
Provisioned Read Capacity Units	Number of read capacity units you provisioned for a table (or a global secondary index) during the selected time period .
Provisioned Write Capacity Units	Number of write capacity units you provisioned for a table (or a global secondary index) during the selected time period .

5.3 Speed and throughput

One of the most popular metrics to measure database performance is the speed to perform a full transaction taking into account its throughput, that involves the number of completed transactions in a unit of time. For each pf the following bar charts, the tables showing the exact results are provided in appendix B while the errors charts to check the Standard Deviation is provided in appendix A.

The following benchmarks were performed for read operations setting read units to 200. In DynamoDB *Query* operation is expected to be very fast and only slightly slower than Get operation. The scan operation on the other hand may take anywhere from few milliseconds to a few hours to complete depending on the size of the table because it is performed by going through each item in the table. For any reasonably sized table the scan operation will consume all the provisioned read units until the operation finishes. Indeed, in fig. 5.2 Scan operation is the one that takes most to execute the fetching of the data.

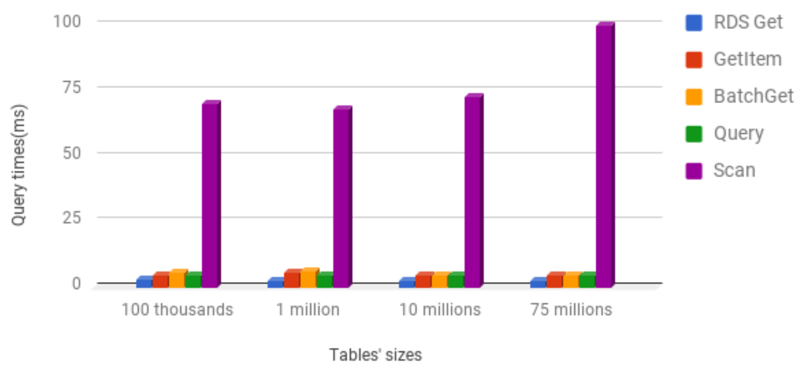


Figure 5.2
Read operations with Dynamo Scan (table B.6)

To have a better overview of the query times, in fig.5.3 it has been decided to get rid of *Scan* operation and shows only the others. It was expected to see DynamoDB being faster than MariaDB in retrieving data, especially with the growth of the tables' sizes. As it is clear, RDS Get is faster on the overall test ranging between 2.26 and 2.86 ms compared with any of the DynamoDB's queries. Query operations results stable overall and mostly faster than both GetItem and BatchGet which seem to perform better with large size tables than smaller ones.

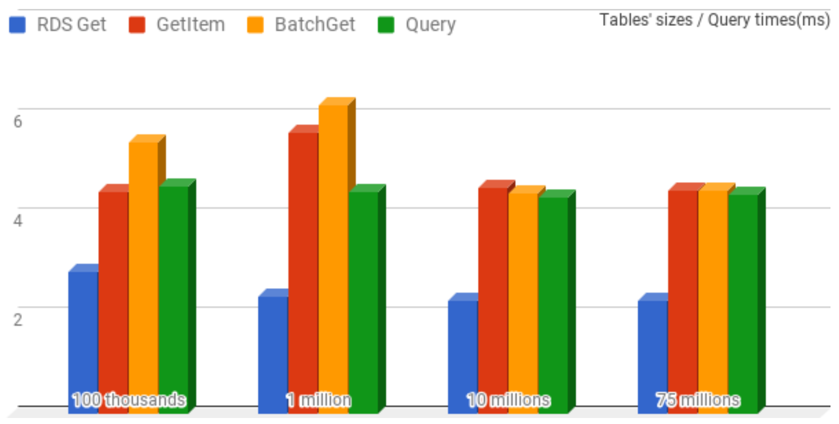


Figure 5.3
Read operations without Dynamo Scan(table B.5)

Write benchmarks described in subsections 4.2.2.1 and 4.2.3.1 with 500 writes units are shown in fig.5.4. Against every expectation the overall write operations run from DynamoDB turns out to be faster than the RDS counterpart. DynamoDB PutItem results faster than SaveRDS which takes more than double of the time to retrieve one item. *DynamoDB BatchWrite* with a range of 33 to 37 ms still performs faster inserts with batch of 25 items than *RDS BulkCreate* that takes from 84*ms* in the smallest table up to 115*ms* in the biggest one. This test also shows the scalability performed on PutItem by DynamoDB while other operations seem to be affected by the increase in size of the table.

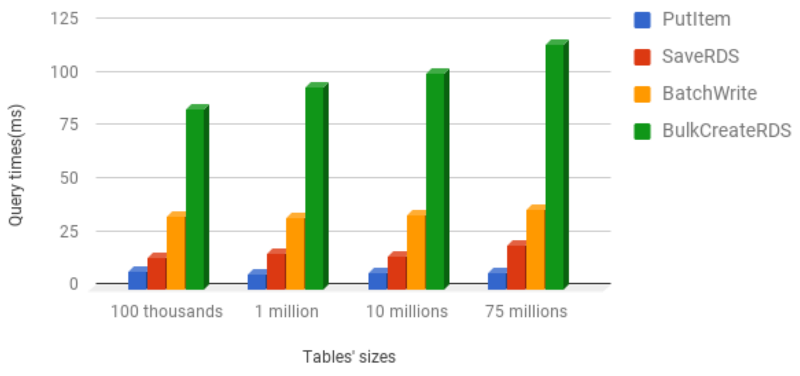


Figure 5.4
Write operations(table B.2)

Fig.5.5 shows the comparison between two updates operations described in subsections 4.2.2.2 and 4.2.3.2. Both write and read capacity units were set to 100. Throughout the test all the update operations made by RDS remain balanced around $4ms$ and faster than DynamoDB update operations that are also pretty balanced around $5ms$ with a peak of almost $6ms$ in the $75millions$ items table.

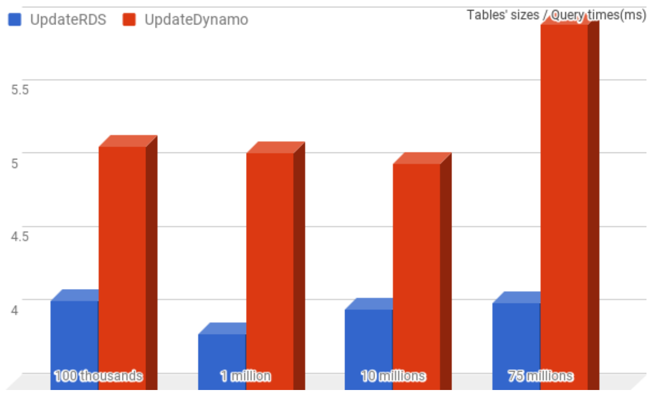


Figure 5.5
Update operations(table B.3)

As shown in fig.5.6 delete one item from RDS MariaDB employs on average around $4ms$ resulting slightly faster than DynamoDB. DeleteItemDynamo(code 4.2.3.3) seems to be affected by the increasing table size, indeed, it takes around $5ms$ towards the $100thousands$ table ms and the query times rises up to around $6ms$ for the item's deletion from the biggest table of $75millions$ items. Besides single item's deletion, a 25 items' deletion from their primary keys was tested in DynamoDB while MariaDB does not own any query to delete more keys in parallel. BatchDeleteDynamo has a peak at 8.44ms in the smaller table while the query time seems to balance at around 5ms in the other tables being even faster than RDS in the table with 10$millions$ items. All the delete operations from DynamoDB had read capacity units set to 100.

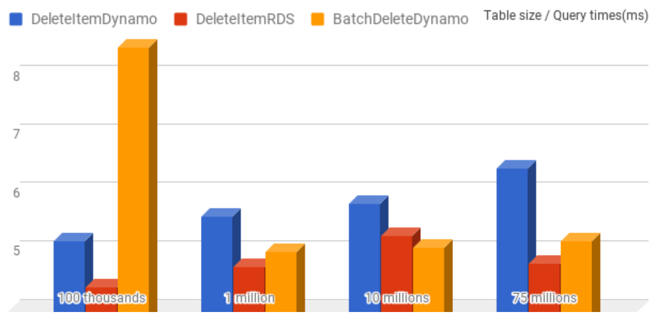


Figure 5.6
Delete operations(table B.4)

Fig. 5.7 shows the query times to retrieve items matching a provided index key-value(in code 4.2.3.5 and 4.2.2.5) while fig. 5.8 shows query times to fetch items by providing index key-value and conditions(in code 4.2.3.5 and 4.2.2.5). Both operations from DynamoDB had read capacity units set to 100. The results from both the tests look pretty similar meaning that with this implementation, querying a table towards an index with or without conditions does not affect the query time too much. However, the difference in time among the operation from RDS and DynamoDB in both the test is embarrassedly huge with RDS constantly standing between 0.28 and 0.42 milliseconds while DynamoDB ranging between roughly 900 and 921 milliseconds in fig. 5.7.

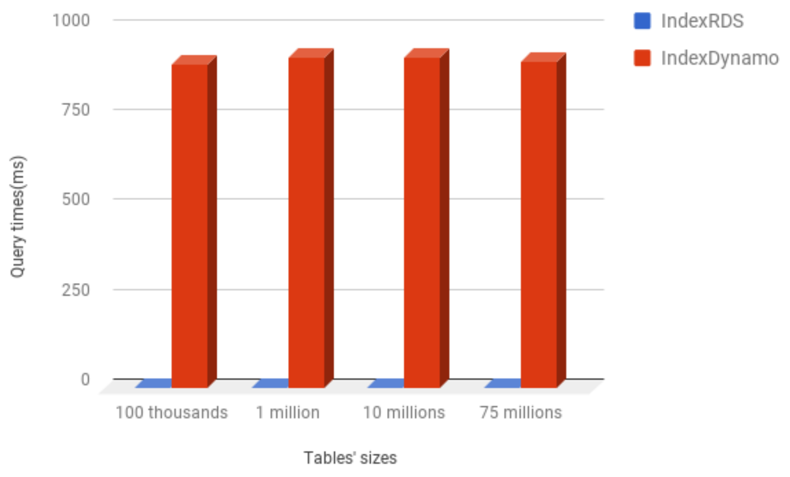


Figure 5.7
Query on index operations(table B.7)

Fig. 5.8 shows that results from DynamoDB range a bit better than the ones in fig. 5.7 scoring times between 627 and 671 milliseconds but still way higher than RDS which maintains the boundaries between around 0.44 and 0.57 milliseconds.

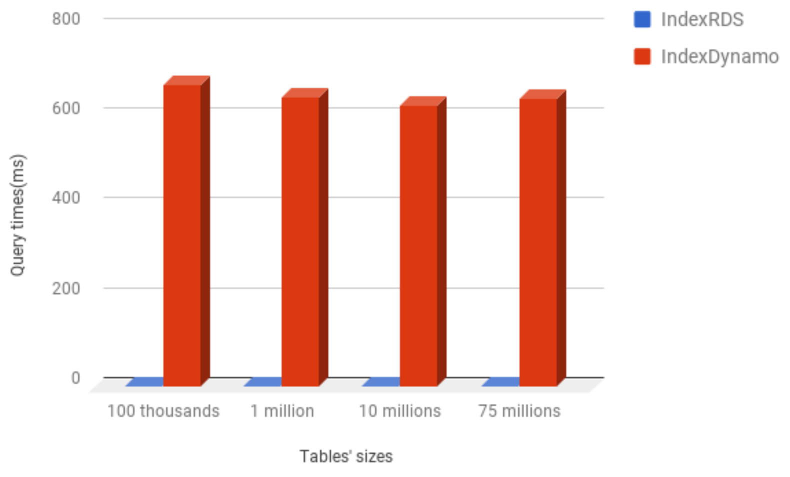


Figure 5.8
Query on index with conditions(table B.8)

Due to the complexity of the dyanamoDb system, the obtained operational times are very much the expected behavior of the DynamoDB system. DynamoDb being a noSQL database, owns a lot of moving parts on the backend. What has been mainly experienced throughout the analysis is that at any time, a given request could vary in times due to the network traffic/latency on Amazon DynamoDB backend nodes. Being present numerous components on a network, such as DNS servers, switches, load balancers, and others can generate delays anywhere in the life of a given request due to which it is not possible to provide an exact operational time for DyanmoDb requests.

Dynamodb only promises to customers in average a single digit millisecond latency on a single item API calls such as *getItem, updateItem, DeleteItem and putItem* only when the item size is less than $4KB$ for reads and $1KB$ for writes. For batches operations such as *query* and *Scan*, since the number of results returned by respective API calls varies due to which it is difficult to provide any precise latency values.

DynamoDB latency promises are only made for the server side latency i.e. it is monitored from the moment the requests hits the DynamoDB end-point till it fetches the result, and sends it back to the end-point. Any delays/latencies occurred during the time request reaches from client instance to the end-point or return is not calculated in the DynamoDB latency.

In our case, the latencies obtained might be more than the DynamoDB latency promises since it involves the client side latency as well, or they might have been minor fluctuations from the service side.

5.4 Scalability

Even if DynamoDB does not have any limit on the amount of data being stored in a table, while RDS fixes the limit at 6TB per table, under the scalability point of view, from all the bar charts exposed so far it could be admitted that both databases guarantee high scalability. On one hand, DynamoDB scales automatically spreading the data over sufficient machine resources to meet storage requirements. On the other hand, RDS allows to set from 1,000 IOPS to 30,000 IOPS per DB Instance for scalability reasons. Both databases are able to maintain roughly the same query time with the increase in size of the tables performing the exactly the same operation.

Chapter 6

Conclusions and future work

6.1 Conclusions

Nowadays, web application can generate and consume huge amounts of data. They might start with just a few users and grow drastically to millions creating thousands of write/read operations per seconds and generating a large amount of gigabytes(if not terabytes) per day. Due to this, scalability is not an option anymore, it is a must.

At this time, new RDBMSs providing improved horizontal scaling have been introduced in the market, however, the SQL systems still strive to provide horizontal scalability without getting rid of SQL itself and ACID transactions. This fact represents one of the main reasons why many companies are moving their database layers to a NoSQL solution. Subsection 2.6.1 presented several types of available NoSQL solutions describing some of their key features. The right one to chose strictly depends on company's use cases. In general talking, if the application needs to handle a lot of data, it will be limited when using MySQL or other RDBMS by their structure. The right choice between NoSQL and RDBMS only depends on the actual needs of the application. In fact, some applications might be well served by using both. On one hand, if the application require to store data that does not lend itself well to a relational schema (tree structures, schema-less JSON representations, and so on) that can be looked up against a single key or a key/range combination then DynamoDB (or some other NoSQL store) would likely be the best bet. On the other hand, to have a well-defined schema for application's data that can fit well in a relational structure and the flexibility to query the data in a number of different ways (adding indexes as necessary of course) is strictly required, then RDS might be a better solution(answer to **RQ1**). Our choice in respect of Amazon DynamoDB was guided by the structure of the dataset in the tables subjects of the analysis. Such dataset could be easily interpreted as a key-value and since the system's migration was about to be done in favor of AWS, DynamoDB has been chosen for the purpose of the benchmarking analysis.

In the overall considerations, DynamoDB results highly available and stable against failures thanks to the automatic replication and failover policies. It shows an excellent scalability and aggregated throughput, although, it guarantees the maximum throughput instead of latency, users will not get faster response if they use low throughput. This means that with a large amount of users it will not work as fast as it would with fewer users, instead, with few users it will work as slow as with many. This ”feature” has been built to avoid users from reaching easily the provisioned throughput by lowering down the latency in the case they have only fewer users [50].

RDS is also a Managed service, however the use case for both of them is entirely different. Thus, query speeds in DynamoDB and RDS also depends upon a lot of different factors as explained in chapter 5. DynamoDB as NoSQL storage layer is mainly considered for lookup queries (and not Join queries).

Our benchmarking analysis exposed in chapter 5 shows that choosing a NoSQL database to speed up query times is not always the right chose to be done(at least with our implementation and tables' structure). In our case, DynamoDB have not met our expectation. We expected to see its query times to be faster in fetching records than writing them into the tables, thus, compared with Amazon RDS, most of the queries performed were slower than the latter with the exception of queries related to write operations where DynamoDB unexpectedly jumped up in speed resulting overall faster than RDS. The main benefit for using DynamoDB as a NoSQL store is that the user gets guaranteed read/write throughput at whatever level required without having to worry about managing a clustered data store. If, for example, the application requires a thousand reads/writes per second, DynamoDB table can just be provisioned for that level of throughput and the developer does not really have to worry about the underlying infrastructure. Moreover, it allows the user to start small and arrange capacity's table as long as the requirements increase without being exposed to downtimes(research question **RQ2**).

Having said that and looking at the overall picture under another point of view, taking care of database scalability, management, performance, and reliability does not cover all the lacks from the functionalities that can be found in a relational database. In DynamoDB querying data is extremely limited especially about non-indexed data. Not to have a good support for complex transactions or complex queries like JOINs could be a big concern for a workload that requires this range of capabilities even if SQL JOIN operations, for example, kill performance, especially when is required to aggregate data across such joins. Thinking at the situation where there are dozens of joins with thousands of simultaneous users, an RDBMS system would start to fall apart. On the other hand, in NoSQL databases complex data relations have to be managed on the code/cache layer and sometimes it can be hard to do. Furthermore, on matter of data integrity, while the idea of fluid data may be interesting and desirable to begin with, some applications' workloads may be better suited if used with an unchangeable structure. Strong type might save the whole database when a little bug attempts to destroy it and in DynamoDB, being schema-less, most of the time anything that can go wrong, it exactly does(research question **RQ3**).

Moreover, as explained from Eric Brewer in [42], a system could maintain only two out of three among consistency, availability and partition-tolerance and in case of NoSQL systems, generally, consistency is the one that gives up. Most of the time, the lack of consistency brings to better performance and scalability but for some kinds of applications and transactions, like the ones involved on banking services, it could be a real issue(research question **RQ4**).

One more concern to take into account is the limit in matter of indexes in DynamoDB. Changing or adding Local Secondary Indexes on-the-fly is impossible without creating a new table. Global Secondary Indexes creation is available up to five meaning that if more indexes are required workarounds have to be implemented in the code. Moreover, once the read/write capacity units are set, Amazon allows to decrease them only four times per day while there is no limits of provisioned IOPS in RDS for MariaDB.

Eventually, NoSQL databases are targeted to a specific set of applications scenarios. They will not and cannot replace RDBMSs, they can address certain limitations and are a complement to deal with issues such as complexity, performance, and scalability. Once again, the right database to be used should be chosen according to the data model and dataset to be used with.

6.2 Future work

As this work aimed to analyze the performance testing with just simple queries, there are several possibilities for further extensions and improvements. One of the most interesting aspects that could be explored would be the behaviour of DynamoDB in presence of queries involving multiple tables with JOINs operations. This would require a precise schema to be implemented, for example, involving redundancy of data, to test if effectively a NoSQL database can deal with lack of JOIN's operator.

Another interesting topic for future works could be the implementation of our benchmarking analysis on other NoSQL databases besides DynamoDB, like document stores or column stores presented in subsections 2.6.1.2 and 2.6.1.4. This would give the opportunity to compare the performance of them with real world data.

Other further investigations about performance improvements could be made by adding caching, that in key-value stores seems to be an excellent use case when running a medium to high volume dataset where data is mostly read.

Moreover, the benchmarking analysis involved in this project could also be replicated in other clouds solutions like OpenNebula [58] or Microsoft Azure [53] as long as they provides services for NoSQL databases with similar features of the ones own by Amazon DynamoDB.

Bibliography

[1] Replicated data management in distributed systems. 1992.

[2] Object-relational dbmss: The next great wave. *IBM Redbooks*, 1996.

[3] Ims premier. *IBM Redbooks*, 2000.

[4] *Database Management Systems 3nd edition.* McGraw-Hill Higher Education, 2003.

[5] Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[6] Cassandra: A decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, April 2010.

[7] *Database System Concepts 6th Edition.* McGraw-Hill, January 2010.

[8] Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):20–27, August 2012.

[9] Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4), March 2013.

[10] Benchmarking couchbase for interactive applications with high in-memory data loads. March 2014.

[11] Relational vs. nosql databases: A survey. *International Journal of Computer and Information Technology*, 03(3):598–601, May 2014.

[12] Database migration from structured database to non-structured database. *International Journal of Computer Applications (ICRTAET 2015)*, 2015.

[13] Nosql databases: a survey and decision guidance. `https://medium.baqend.com/`, August 2016.

[14] Sql versus nosql movement with big data analytics. pages 59–66, December 2016.

[15] Geeta Pattun Abdul Haseeb. A review on nosql: Applications and challenges. *International Journal of Advanced Research in Computer Science*, 8(1), Jan-Feb 2017.

[16] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Which nosql database? a performance overview. *Open Journal of Databases(OJDB)*, 1(2), 2014.

[17] Inc. Amazon Web Services. Amazon aws. `https://aws.amazon.com/`.

[18] Inc. Amazon Web Services. Amazon ec2. `https://aws.amazon.com/ec2/?nc2=h_m1`.

[19] Inc. Amazon Web Services. Boto3. `https://boto3.readthedocs.io`.

[20] Inc. Amazon Web Services. Amazon dynamodb. `http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/Welcome.html`, 2017.

[21] Inc. Amazon Web Services. Local secondary indexes. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html`, 2017.

[22] Inc. Amazon Web Services. Paginating the results. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html#Query.Pagination`, 2017.

[23] Inc. Amazon Web Services. Partitions and data distribution. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html`, 2017.

[24] Charles W. Bachman. The programmer as navigator. *Communications of the ACM*, 16(11):653–658, November 1973.

[25] Alan Beaulieu. *Learning SQL, 2nd Edition.* O'REILLY, April 2009.

[26] Paul Beynon-Davies. *Database Systems 3rd edition.* PALGRAVE MACMILLAN, 2004.

[27] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. Mongodb vs oracle – database comparison. pages 330–335, 2012.

[28] E. F. Codd. A relational model of data for large shared data banks. 1970.

[29] E. F. Codd. Further normalization of the data base relational model. August 1971.

[30] E. F. Codd. Recent investigations into relational data base systems. April 1974.

[31] E. F. Codd. *The relational model for database management: version 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[32] Oracle Corporation. How mysql uses indexes. `https://dev.mysql.com/doc/refman/5.7/en/mysql-indexes.html`, 2017.

[33] Tanmay Deshpande. *Mastering DynamoDB.* PACKT, August 2014.

[34] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems.* Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[35] Django Software Foundation. Django 1.9.4. `https://www.djangoproject.com`.

[36] Django Software Foundation and individual contributors. `https://docs.djangoproject.com/en/1.11/ref/models/instances/`, 2005-2017.

[37] Python Software Foundation. socket — low-level networking interface. `https://docs.python.org/2/library/socket.html`, 1990-2017.

[38] Python Software Foundation. Time access and conversions. `https://docs.python.org/2/library/time.html`, 2017.

[39] The Apache Software Foundation. Cassandra. `http://cassandra.apache.org/`.

[40] The Apache Software Foundation. Couchdb. `http://couchdb.apache.org/`.

[41] Shahram Ghandeharizadeh, Jason Yap, and Hieu Nguyen. Strong consistency in cache augmented sql systems. pages 181–192, 2014.

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

[43] Github. Titan. `http://titan.thinkaurelius.com/`.

[44] Venkat N. Gudivada, Dhana Rao, and Vijay V. Raghavan. Nosql systems for big data management. pages 190–197, 2014.

[45] Amazon Web Services Inc. Amazon dynamodb. `https://aws.amazon.com/documentation/dynamodb/`.

[46] Amazon Web Services Inc. Amazon relational database service. `http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`.

[47] Maria Indrawan-Santiago. Database research: Are we at a crossroad? reflection on nosql. pages 45–51, 2012.

[48] Charles Kozierok. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.* No Starch Press, San Francisco, CA, USA, 2005.

[49] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, February 2010.

[50] Tonglin Li, Xiaobing Zhou, Kevin Br, and Ioan Raicu. Distributed key-value store on hpc and cloud systems, 2013.

[51] LinkedIn. Project voldemort. `http://www.project-voldemort.com/`.

[52] MariaDB. About xtradb. `https://mariadb.com/kb/en/mariadb/about-xtradb/`, 2017.

[53] Microsoft. Microsoft azure. `https://azure.microsoft.com`, 2017.

[54] Inc. MongoDB. Mongodb. `https://www.mongodb.com`.

[55] Inc. Neo Technology. Neo4j. `https://neo4j.com/`.

[56] Jaroslav Pokorny. Nosql databases: A step to database scalability in web environment. pages 278–283, 2011.

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

[58] OpenNebula Project. Opennebula. `https://opennebula.org/`, 2017.

[59] M. R. Patra R. P. Padhy and S. C. Satapathy. Rdbms to nosql: Reviewing some next-generation non-relational database's. *International Journal of Advances in Engineering, Science and Technology*, 11(1015):15–30, 2011.

[60] REDISLABS. Redis. `https://redis.io/`.

[61] Stuart Sechrest. An introductory 4.3bsd interprocess communication tutorial. *Unix Programmer's Supplementary Documents*, 1, April 1986.

[62] Werner Vogels. Eventually consistent. 2008.

[63] Adrienne Watt. *Database Design.* B.C. Open Textbook project, October 2012.

Appendix A

Standard Deviation on measurements

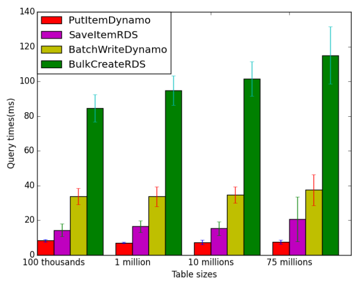


Figure A.1
Standard Deviation on Write operations

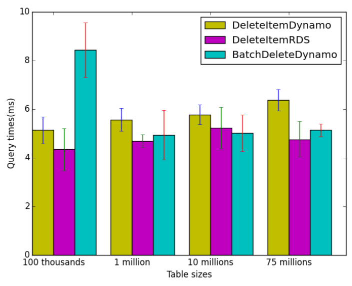


Figure A.2
Standard Deviation on Delete operations

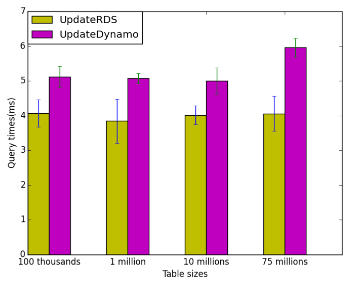


Figure A.3
Standard Deviation on Update operations

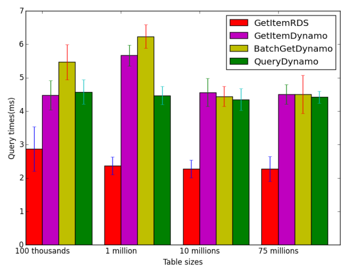


Figure A.4
Standard Deviation on Read operations

Appendix B

Measurement Tables

Table Size	Test 1	Test 2	Test 3	Test 4	Test 5
Connection	33.829152	1.904969	1.882047	1.886137	1.923552
TCP ping	0.606203	1.139283	0.637483	1.196694	1.185750
RDS Get	2.193022	2.171395	2.130826	2.112163	2.111112
Dynamo Get	5.934749	4.802551	5.147996	5.105661	5.329197
Dynamo Batch	6.123057	4.939521	5.237638	4.959187	6.658048
Query	6.179594	4.90613	5.117563	4.775432	5.834731
Scan	6.092347	33.379617	33.311775	33.338368	33.346722

Table B.1: Connection times5.1

Table Size	PutItem	SaveRDS	BatchWrite	BulkCreateRDS
100 thousands	8.35632	14.3593	33.84644032	84.5713289
1 million	7.08584	16.4825	33.72320054	94.84135521
10 millions	7.35798	15.3919	34.69473784	101.4228945
75 millions	7.6677	20.6777	37.5436328	115.0309117

Table B.2: Query times of write operations5.4

Table Size	UpdateRDS	UpdateDynamo
100 thousands	4.072516951	5.120555672
1 million	3.845012568	5.077892324
10 millions	4.014368688	5.003209089
75 millions	4.056824424	5.957543907

Table B.3: Query times of update operations5.5

Table Size	DeleteItemDynamo	DeleteItemRDS	BatchDeleteDynamo
100 thousands	5.13576	4.346059582	8.441925049
1 million	5.5659	4.687962605	4.945039749
10 millions	5.7765	5.232647829	5.028963089
75 millions	6.37653	4.749733263	5.138158798

Table B.4: Query times of delete operations5.6

Table Size	RDS Get	GetItem	BatchGet	Query
100 thousands	2.86985	4.47293	5.47253	4.57457
1 million	2.36627	5.66778	6.23182	4.47119
10 millions	2.27676	4.55873	4.44165	4.349
75 millions	2.2687	4.50455	4.50312	4.42043

Table B.5: Query times of read operations5.3

Table Size	RDS Get	GetItem	BatchGet	Query	Scan
100 thousands	2.86985	4.47293	5.47253	4.57457	70.2685
1 million	2.36627	5.66778	6.23182	4.47119	67.9173
10 millions	2.27676	4.55873	4.44165	4.349	72.5129
75 millions	2.2687	4.50455	4.50312	4.42043	99.7251

Table B.6: Read operation measurements with Scan5.2

Table Size	IndexRDS	IndexDynamo
100 thousands	0.42135403	900.55684210
1 million	0.28062441	921.59544002
10 millions	0.28042024	921.41932658
75 millions	0.29035418	908.36303211

Table B.7: Query times of read operations on index5.7

Table Size	IndexRDS	IndexDynamo
100 thousands	0.57958679	671.15132658
1 million	0.49703823	644.91465506
10 millions	0.45861939	627.17970174
75 millions	0.44604116	642.31136806

Table B.8: Query times of read operations on index with conditions5.8