Faculty of Sciences and Technology
Department of Informatics Engineering

CBench-Dynamo

A Consistency Benchmark for NoSQL Database Systems

Miguel Prata Leal Branco Diogo

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Bruno Cabral and Prof. Jorge Bernardino and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2019

This page is intentionally left blank.

Acknowledgements

I would first like to thank my thesis advisors Prof. Bruno Cabral and Prof. Jorge Bernardino for their patience and perseverance.

Also, I want to thank all my friends for cheering for me.

Finally, I want to thank my parents and sister for having shaped my values and believes towards the person I am today.

This page is intentionally left blank.

Abstract

Nowadays software architects face new challenges. The Internet has grown to a point where popular websites are accessed by hundreds of millions of people on a daily basis. One powerful machine is no longer economically viable and resilient in order to handle such outstanding traffic. Architectures have since been migrated to horizontal scaling. However, traditional databases, usually associated with a relational design, were not ready for horizontal scaling. Therefore, NoSQL databases have proposed to fill the gap left by their predecessors. This new paradigm is proposed to better serve currently massive scaled-up Internet usage when consistency is no longer a top priority and a high available service is preferable. However, based on the CAP theorem when in a distributed environment where network partition events occur, only one of the two properties, consistency or availability, can be guaranteed. When one increases the other must decreases. Dynamo-based databases are designed to run in a cluster while offering high availability and eventual consistency to clients when subject to network partition events. Therefore, this thesis proposes CBench-Dynamo, the first consistency benchmark for NoSQL databases. The proposed benchmark correlates properties, such as performance, consistency, and availability, in different consistency configurations while subjecting the System Under Test (SUT) to network partition events. This enables us to better comprehend how the SUT handles the trade-off between these properties.

Keywords

Consistency, Availability, Network Fault Tolerance, NoSQL Databases, Benchmark, Dynamo, Cassandra.

This page is intentionally left blank.

Resumo

Hoje em dia arquiteturas de software encaram novos desafios. A Internet cresceu tal que existem sítios na Internet que são acedidos por centenas de milhões de pessoas diariamente. Uma única máquina poderosa não é mais economicamente viável e resiliente de forma a lidar com a imensidão de tráfego e as arquiteturas têm desde então sido migradas para escalagem horizontal. No entanto, bases de dados tradicionais, mais associadas ao paradigma relacional, não estão preparadas para a escalagem horizontal. Desta feita, as base de dados NoSQL vieram propôr preencher essa limitação. O paradigma NoSQL propõe melhor servir a atual massificação de uma Internet com alto tráfego de dados onde a consistência não é uma prioridade de topo, mas sim a alta disponibilidade para muitos projetos. No entanto, de acordo com o teorema de CAP entre as duas propriedades, disponibilidade ou consistência, só uma delas pode ser totalmente garantida. A especificação Dynamo consiste num cluster de bases de dados que oferecem alta disponibilidade enquanto relaxam a consistência ao nível de consistência eventual ao mesmo tempo que toleram eventos de partição na rede. Consequentemente, esta tese propõe CBench-Dynamo, a primeira framework de benchmark para bases de dados NoSQL. O benchmark proposto correlaciona propriedades, como performance, consistência e disponibilidade, em diferences configurações de consistência enquanto sujeitamos o sistema em testes a eventos de partição na rede. Consequentemente, permitindo-nos compreender melhor como o sistema em testes gere os *trade-offs* entre estas propriedades.

Palavras-Chave

Consistência, Disponibilidade, Tolerância a Partições na Rede, Bases de Dados NoSQL, Benchmark, Dynamo, Cassandra.

This page is intentionally left blank.

Contents

1 Introduction **1**
- 1.1 Thesis Goals . . . 1
- 1.2 Main Contribution . . . 2
- 1.3 Work Recognition and Publication . . . 2
- 1.4 Report Structure . . . 2

2 Background Knowledge **4**
- 2.1 CAP Theorem . . . 4
- 2.2 Consistency Models . . . 4
  - 2.2.1 Strong Consistency . . . 5
  - 2.2.2 Weak Consistency . . . 6
  - 2.2.3 Eventual Consistency . . . 6
  - 2.2.4 Causal Consistency . . . 6
  - 2.2.5 Read-your-writes Consistency . . . 7
  - 2.2.6 Session Consistency . . . 7
  - 2.2.7 Monotonic Reads Consistency . . . 7
  - 2.2.8 Monotonic Writes Consistency . . . 7
- 2.3 Dynamo . . . 7
  - 2.3.1 Consistency Configurations . . . 8
  - 2.3.2 Consistency Faults . . . 9

3 Related Work **12**
- 3.1 NoSQL Background . . . 12
- 3.2 Analytical Models to Study Consistency on NoSQL Databases . . . 14
  - 3.2.1 ALL Write Consistency Level or ALL Read Consistency Level . . . 15
  - 3.2.2 ONE Read Consistency Level and QUORUM Write Consistency Level 15
  - 3.2.3 QUORUM Read Consistency Level and ONE Write Consistency Level 15
  - 3.2.4 ONE Read Consistency Level and ONE Write Consistency Level . . 16
- 3.3 Benchmark Consistency in NoSQL databases . . . 16

4 CBench-Dynamo **20**
- 4.1 CBench-Dynamo Properties . . . 20
  - 4.1.1 Relevance . . . 21
  - 4.1.2 Reproducibility . . . 21
  - 4.1.3 Fairness . . . 22
  - 4.1.4 Verifiability . . . 23
  - 4.1.5 Usability . . . 23
- 4.2 Methodology . . . 23
- 4.3 Architecture Specification . . . 24
  - 4.3.1 Orchestrator . . . 24
  - 4.3.2 Load Balancer . . . 31

4.3.3 Data Analyser . . . 32
4.3.4 Network partition events generator . . . 33

5 Experimental Evaluation **36**
5.1 Cassandra . . . 36
5.2 Architecture . . . 37
5.3 Experiment . . . 37

6 Work Plan **40**
6.1 First Semester . . . 40
6.1.1 Survey . . . 41
6.1.2 Proof of Concept . . . 41
6.2 Second Semester . . . 41
6.2.1 Implementation . . . 41
6.2.2 Conference Paper . . . 41

7 Conclusions and Future Work **42**

This page is intentionally left blank.

Acronyms

PoC Proof of Concept. 42

SUT System Under Test. v, 1, 2, 4, 20, 22, 24, 25, 36, 37, 43

This page is intentionally left blank.

List of Figures

2.1 CAP Theorem [1]. 5
2.2 Data-centric and Client-centric consistencies [2] 5
2.3 Consistency Models based on Reference [3]. 6
2.4 Partitioning and replication of keys in Dynamo ring [4]. 8
2.5 Data request on abnormal operation where a node fails, and strong read consistency is set (i.e. *ALL*). 10
2.6 Data request on abnormal operation where a node fails, and eventual read consistency is set (i.e. *ONE*). 10

3.1 Cassandra ring [5]. 13
3.2 MongoDB replica set example [5]. 13
3.3 Cassandra vs MongoDB [5]. 14
3.4 PBS results for (N = 5, R = 1, W = N = 5) and (N = 5, R = N = 5, W = 1). 15
3.5 PBS results for (N=5, R=1, W=3). 15
3.6 PBS results for (N=5, R=3, W=1). 16
3.7 PBS results for (N=5, R=1, W=1). 16
3.8 Bermback and Tai's propose benchmark architecture. 18

4.1 Dynamo-based databases set within NoSQL realm. 21
4.2 Example of Ansible facts. [6] 22
4.3 Proposed benchmark's results data structure. 24
4.4 CBench-Dynamo Architecture Specification. 24
4.5 Cassandra nodes and coordinator instances listed on AWS Management Console. 25
4.6 Ansible generic playbook. 25
4.7 Ansible hosts file. 26
4.8 Ansible generic role. 26
4.9 Ansible playbook for setting up and preparing for testing a Cassandra node in a AWS EC2 instance. 27
4.10 Ansible role for setting up a Cassandra node in a AWS EC2 instance. 28
4.11 Ansible playbook for preparing a Cassandra node in a AWS EC2 instance. . . 28
4.12 Ansible playbook for setting up the benchmark coordinator. 29
4.13 Ansible role for setting up the benchmark coordinator. 29
4.14 Ansible playbook for running a set of benchmark configurations. 30
4.15 Ansible role for running a single benchmark configuration. 31
4.16 Load balancer instance used listed on AWS Management Console. 32
4.17 Ansible role for analysing data. 32
4.18 Main logic to calculate the measurements evaluated. 33
4.19 Ansible playbook for initiating the network partition event generator. 34
4.20 Ansible role that randomly chooses the cluster node to fail and calls the network partition event bash script. 34

4.21 Bash script that stops Cassandra service on the cluster node received as argument, sleeps for specific time, and recovers the cluster node back to life. 35
4.22 Network partition events generator output. 35

5.1 Testing architecture. 37
5.2 Consistency and availability results for all configurations. 39
5.3 Write latency and read latency results for all configurations. 39

6.1 Diagram Gantt. 40

This page is intentionally left blank.

List of Tables

3.1 Related work summary. 19

4.1 Workload configurations. 30

5.1 Cassandra's benchmark workload configurations. 38
5.2 Availability and nines notation [7] 38

This page is intentionally left blank.

Chapter 1

Introduction

The Internet has grown to a point where billions of people have access to it, not only from a desktop but also from smartphones, smartwatches, and even other servers and services. Nowadays systems need to scale. The monolithic database architecture, based on a powerful server, does not guarantee the high availability and network partition tolerance required by today's web-scale systems, as demonstrated by the CAP (Consistency, Availability, and Network Partition Tolerance) theorem [8]. Strong consistency is a property that has been relaxed to achieve more scalable database systems. Relational databases were designed to support strong consistency. Each transaction must be immediately committed, and all clients will operate over consistent data states. Reads from the same object will present the same value to all client requests. Although strong consistency is the ideal requirement for a database, it deeply compromises horizontal-scalability. Horizontal scalability is a more affordable approach when compared to vertical scalability. It enables higher throughput and data distribution across multiple database nodes. On the other hand, vertical scalability relies on a single powerful database server to store data and answer all requests. Although horizontal scaling may seem preferable, CAP theorem states that when network partitions occur, one has to opt between availability and consistency [9]. Horizontal scaling has inspired a new category of databases called NoSQL. These systems have been created with a common requirement in mind, scalability. Several NoSQL designs prioritize high-availability over a more relaxed consistency strategy, an approach known as BASE (Basically Available, Soft-state and Eventually consistent) [10].

Although frameworks, such as YCSB [11], have been developed for benchmarking NoSQL databases, they lack a consistency tier to fully compare the tradeoffs described by the CAP theorem.

This thesis proposes CBench-Dynamo, a benchmark for testing consistency and availability on a horizontal-scaled system. It also defines the main quality attributes of a benchmark, i.e. Relevance, Reproducibility, Fairness, Verifiability, and Usability [12]. This thesis' main goal is to propose a consistency benchmark framework and extract different measurements on performance, consistency, and availability with different consistency configurations of the System Under Test (SUT) while subjecting this system to network partition events.

1.1 Thesis Goals

This thesis aims to build a NoSQL Consistency Benchmarking Framework that compares and analyses how consistency is affected by other quality attributes such as availability and

performance when subjecting the target system to network partition events. This thesis also aims to take advantage of this new framework and test it on Cassandra. Cassandra can be configured to be Eventual Consistent or Strong Consistent as they implement the dynamo blueprints first introduced by Amazon. This work will enable Software Architects to better assert what NoSQL database will better suit their application requirements based on different consistency configurations and how it influences performance and availability while subjecting the system to network partition events.

1.2 Main Contribution

This study differentiates from all the work in this field by purposing the first NoSQL consistency benchmark while subjecting the SUT to network partition events. This work also tests the proposed benchmark on Cassandra.

To the best of our knowledge, there is no other work that empirical compares the three vertices of the CAP theorem, consistency, availability and network partition tolerance.

This thesis extends the work done by Bermbach and Tai [13] on long-term benchmarking on S3 by introducing network partition events and a framework approach that can be used on dynamo-based databases such as Cassandra.

All the codebase is openly available on GitHub [14] [15] [16].

1.3 Work Recognition and Publication

This thesis has resulted in two papers, hence contributing to the scientific community within this field. Future Internet published a resulting survey from this thesis work [17] (see Appendix B) about consistency models on NoSQL databases and served as a theoretical introduction to the proposed benchmark framework, CBench-Dynamo, published (see Appendix C) and presented at TPCTC/VLDB 2019 conference in Los Angeles, USA (see presentation in Appendix D).

1.4 Report Structure

The remainder of this report is structured as follows. Section 2 presents the background knowledge. Section 3 presents the related work. Section 4 presents the proposed benchmark framework, CBench-Dynamo, followed by the testing results in Section 5. In Section 6, we present the work plan. Finally, in Section 7, the current thesis' conclusion and future work.

This page is intentionally left blank.

Chapter 2

Background Knowledge

In this chapter, the main concepts are introduced. Section 2.1 presents the CAP theorem as the main theorem that supports our problem and motivation. Then, Section 2.2 gives a brief explanation about the different consistency models in the NoSQL databases realm. Finally, in Section 2.3 we introduce the Dynamo design first introduced by Amazon. This specification will allow us to better understand Eventual Consistency and Availability. This is the design that our System Under Test (SUT) is based on for the proposed benchmark.

2.1 CAP Theorem

In 2000, Eric Brewer introduced CAP theorem [8], Consistency, Availability, network partition tolerance. The CAP theorem states that in a distributed system three properties cannot be fulfilled simultaneously, but only two. According with CAP a preliminary classification of NoSQL databases presents as follows [1]:

- CA (Concerned about consistency and availability). The database's bottom priority is network partition tolerance, and it uses replication as the main approach to ensure data consistency and availability. Traditional relational database are CA.
- CP (Concerned about consistency and partition tolerance). The database's bottom priority is availability. Such database stores data in distributed nodes, while ensuring the consistency of these data. Examples of this configuration are MongoDB (document-based), and Redis (key-value).
- AP (Concerned about availability and partition tolerance). The database's bottom priority is consistency. Such database ensure availability and partition tolerance primarily while relaxing consistency. AP systems are Voldemort (key-value), Riak (document-based), and Cassandra (column-based)

2.2 Consistency Models

In the past, almost all architectures of databases systems were strong consistent. In these cases, most architectures would have a single database instance only responding to a few hundred clients. Nowadays, many systems are accessed by hundreds of thousands of clients, so there was the need to scale database architectures. However, considering the CAP

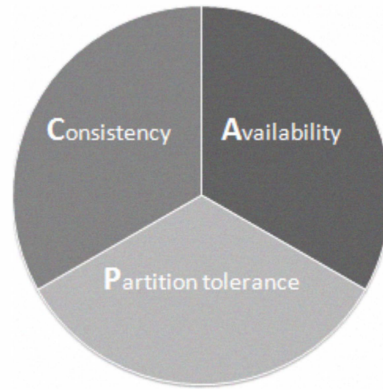


Figure 2.1: CAP Theorem [1].

theorem, high-availability and consistency do conflict on distributed systems when subject to a network partition event. The majority of the projects that have been experiencing such high-traffic have chosen to adopt high-availability over a strong consistent architecture by relaxing the consistency level.

There are two perspectives on consistency, the data-centric consistency and the client-centric consistency, as illustrated in Figure 2.2. Data-centric consistency is the consistency analyzed from the replicas' point of view. Client-centric consistency is the consistency analyzed from the clients' point of view [2].

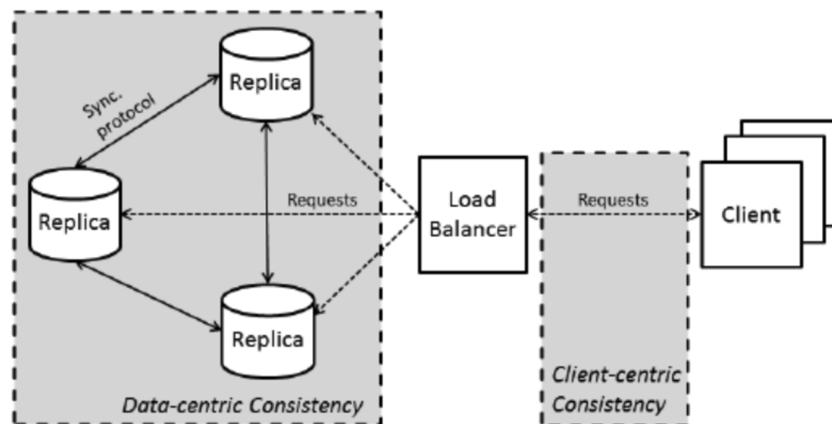


Figure 2.2: Data-centric and Client-centric consistencies [2]

In the next sections, we will review the main consistency models implemented in storage systems: Strong consistency, weak consistency, eventual consistency, causal consistency, read-your-writes consistency, session consistency, monotonic reads consistency, and monotonic writes consistency. Figure 2.3 synthesizes these consistency models.

2.2.1 Strong Consistency

Strong Consistency or Linearization is the strongest consistency model. Each operation must appear committed immediately, and all clients will operate over the same data state. A read operation in an object must wait until the write commits before being able to read the new version. There is also a single global order of events accepted by all storage

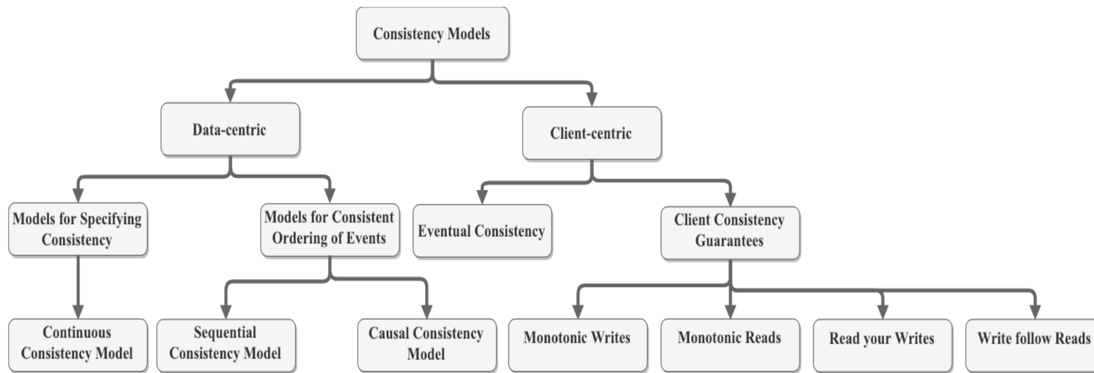


Figure 2.3: Consistency Models based on Reference [3].

systems' instances [18]. Strong Consistency leads to a high consistency system, but it compromises scaling by decreasing availability and network partition tolerance.

2.2.2 Weak Consistency

As the name implies, this model weakens the consistency. It states that a read operation does not guarantee the return of the latest value written. It also does not guarantee a specific order of events [18]. The time period between the write operation and the moment that every read operation returns the updated value is called the inconsistency window [19]. This model leads to a highly scalable system because there is no need to involve more than one replica or node into a client request.

2.2.3 Eventual Consistency

Eventual Consistency strengths the Weak Consistency model. Replicas tend to converge to the same data state. While this convergence process runs, it is possible for read operations to retrieve an older version instead of the latest one. The inconsistency window will depend on communication delays between replicas and its sources, the load on the system and the number of replicas involved [19].

This model is half-way a strong consistency model and a weak consistency model. Eventual Consistency is a popular feature offered by many NoSQL databases. Cassandra is one of them, and it can offer availability and network partition on such a level that it does not compromise the usability of the most accessed websites in the world that uses Cassandra. One of them is Facebook, the company that initially developed Cassandra.

2.2.4 Causal Consistency

If some database client updates a given object, all the clients that acknowledge the update on this object will consider the updated value. However, if some other client (e.g. other database) does not acknowledge the write operation, they will follow the eventual consistency model [19]. Causal consistency is weaker than sequential consistency but stronger than eventual consistency.

Strengthening the Eventual Consistency model to be Causal Consistency decreases availability and network partitioning properties of the system.

2.2.5 Read-your-writes Consistency

Read-your-writes consistency allows ensuring that a replica is at least current enough to have the changes made by a specific transaction. Because transactions are applied serially, by ensuring a replica has a specific commit applied to it, we know that all transaction commits occurring prior to the specified transaction have also been applied to the replica. If some database client updates a given object, this same database client will always consider the updated value. Other clients will eventually read the updated value. Therefore, read-your-writes consistency is achieved when the system guarantees that, once a record has been updated, any attempt to read the record will return the updated value [20].

2.2.6 Session Consistency

If some database client makes a request to the storage system in the context of a session, it will follow a read-your-writes consistency model as long as this session exists. Using session consistency, all reads are current with writes from that session, but writes from other sessions may lag. Data from other sessions come in the correct order, just isn't guaranteed to be current. This provides good performance and good availability at half the cost of strong consistency [21].

2.2.7 Monotonic Reads Consistency

After a database client reads some value, all the successive reads will return that same value or a more recent one [22]. In other words, all the reads on the same object by the same database client follow a monotonic order. However, this does not guarantee monotonic ordering on the read operations between different clients on the same object. Therefore, monotonic reads ensure that if a client performs read r_1, then r_2, then r_2 cannot observe a state prior to the writes which were reflected in r_1; intuitively, reads cannot go backward. Monotonic reads do not apply to operations performed by different clients, only reads by the same client. Monotonic reads can be totally available: Even during a network partition, all nodes can make progress [23].

2.2.8 Monotonic Writes Consistency

A write operation invoked by a database client on a given object needs to be completed before any subsequent write operation on the same object by the same database client [22]. In other words, all the writes on the same object by the same client follow a monotonic order. However, this does not guarantee monotonic ordering on the write operations between different clients on the same object. Therefore, monotonic writes ensure that if a database client performs write w_1, then w_2, then all clients observe w_1 before w_2. Monotonic writes do not apply to operations performed by different clients, only writes by the same client. Monotonic writes can be totally available: Even during a network partition, all nodes can make progress [24].

2.3 Dynamo

Dynamo design and implementation were first introduced by Amazon as a highly available key-value storage system [4]. Since then, Amazon has built many cloud services imple-

menting this design such as Amazon DynamoDB and Amazon S3.

Dynamo prioritizes eventual consistency, targeted to applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes.

Dynamo was designed to scale incrementally, hence it was designed with a partition mechanism in mind. Dynamo's partition mechanism is based on a consistent hashing to distribute the load across multiple data nodes. The output of this hashing function can be illustrated as a ring as seen in Figure 2.4, in which the highest output wraps around the smallest one. Each node is assigned a random value within the range of the hashing function. To know which node will store a given data value, the correspondent key of this value is hashed. Then, we walk the ring clockwise, from the smallest to the largest number, to find the first node with a position larger than the hashing result.

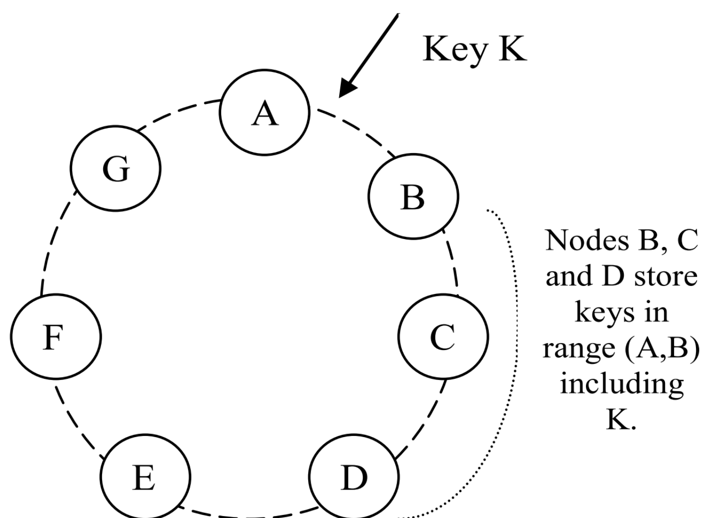


Figure 2.4: Partitioning and replication of keys in Dynamo ring [4].

2.3.1 Consistency Configurations

Amazon's Dynamo based databases such as Cassandra, all use the same variant of quorum-style replication [25]. Quorum-style replication is associated with a replication factor N, i.e. the number of replicas that some data eventual will be in. Read and write consistency can be configured as follows, ONE, $QUORUM$, or ALL.

The following configurations describe the differences between the three write consistency levels for Dynamo-based database systems [17]:

- ALL. Data is written on all replica nodes in the cluster before the coordinator node acknowledges the client. Therefore, this configuration has: Strong Consistency and High latency.
- $QUORUM$. Data is written on a given number of replica nodes in the cluster before the coordinator node acknowledges the client, where this number is called the quorum. This configuration has: Eventual Consistency and Low latency.
- ONE. Data is written in at least one replica node. This configuration has: Eventual Consistency and Low latency.

Analogous to the write consistency levels, the following configuration constants describe the differences between the three read consistency levels for Dynamo-based database systems [17]:

- ALL. The coordinator node returns the requested data to the client only after all replicas have responded. This configuration has: Strong consistency and Less availability.
- $QUORUM$. The coordinator node returns the requested data to the client only after a quorum of replicas has responded. This configuration has: Eventual consistency and high-availability.
- ONE. The coordinator node returns the requested data to the client from the closest replica node. This configuration has: Eventual consistency and High availability.

Under normal operation, i.e. without network partition events, given the number of replicas required for a read operation as R, the number of replicas required for a write operation as W, and the replication factor as N, Dynamo-based databases guarantee consistency when [25]:

$$R + W > N \tag{2.1}$$

Given R_{Quorum} and W_{Quorum}, as Read Consistency and Write Consistency set to *QUORUM*, respectively, and the floor function that takes as input a real number and round it down to the closest integer. The conversion from the *QUORUM* notation to the *R*, *W* notation is as follows:

$$R_{QUORUM},\ W_{QUORUM} = R,\ W = floor\left(\frac{N}{2} + 1\right) \tag{2.2}$$

2.3.2 Consistency Faults

Under abnormal operations where a network partition event had occurred, if consistency is set to strong, i.e. *ALL*, availability is compromised as the Figure 2.5 illustrates.

The scenario illustrated by Figure 2.5 describes a situation where a read operation needs to involve the total number of replicas N in order to retrieve the data to the client. In case of a network partition, e.g. node C crashes, the coordinator of the request, node A, was not able to serve data, hence the total number of replicas had been involved in the operation, and the coordinator had no other option than announcing a lack of service availability to the client, resulting in a $TIMEOUT$ response.

Under abnormal operations where a network partition event had occurred, if consistency is set to eventual configuration (ONE), we achieve service availability even in the presence of a node crash as the Figure 2.6 illustrates. The scenario illustrated by Figure 2.6 describes a situation where a read operation only needs to involve one replica in order to retrieve the requested data to the client. Because only one replica had been involved, the response may not contain the latest data as this example suggests. Although consistent responses are not ensured, this configuration results in a high-available service.

For Dynamo-based databases, high availability does not necessarily ensures write persistence. When addressing the concept of availability, in terms of service availability and not

data availability, i.e. when a request is made, a successful response is given even if such key does not exist anymore. The response is successful, even if the response refers to the inexistent of such resource. There may be the case when, for instance, a configuration of $W = 1$ (ONE) is set and the same node crashes right afterward, the data is lost and there is no acknowledgment to the client that such abnormality had happened. To avoid such events, W values greater than 1 increase data redundancy and, consequently, the probability of all replicas that contain the data fail is diminished.

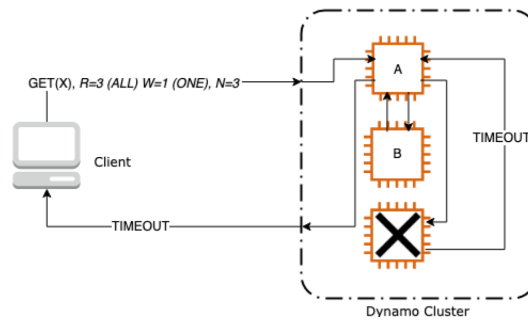


Figure 2.5: Data request on abnormal operation where a node fails, and strong read consistency is set (i.e. ALL).

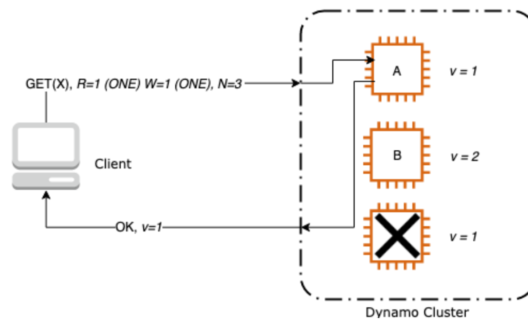


Figure 2.6: Data request on abnormal operation where a node fails, and eventual read consistency is set (i.e. ONE).

Chapter 3

Related Work

This chapter presents the current state of the art related to consistency comparison and benchmarking in NoSQL databases.

3.1 NoSQL Background

Consistency models are analyzed in various works using different assumptions. Bhamra in Reference [5] presents a comparison between the specifications of Cassandra and MongoDB. The author focuses only on a theoretical comparison based on the databases specifications. The objective of this work is to help the reader choosing which database is more suitable for a particular problem. Bhamra starts by making a comparison between Cassandra and MongoDB specifications.

A cluster of nodes in Cassandra is visualized as a ring (see Figure 3.1). Cassandra distributes all the data from the keyspace evenly across all the nodes. Each Cassandra node in the cluster stores a subset of the data and is responsible for an interval of hashes. When some data is inserted into the database, the key of that data is sent to the partitioner. The partitioner is a hash function based on that key. The resulting hash determines which node contains that data's value. Randomness is obtained by using hashing, therefore a fair load is attained across nodes [5]. Data replication is defined per keyspace and follow two parameters; the replication strategy and replication factor. The replication strategy defines the algorithm that decides at which nodes to store copies of rows. In the other hand, the replication factor represents the number of copies of each row have to be persisted.

MongoDB is a schema-less document-oriented database, hence documents are used as the primary structure for persisting data. MongoDB data model is based on documents. These documents are represented in BSON (Binary JSON). This format extends the well-known JSON (JavaScript Object Notation). Documents that represent a similar class of objects are organized as collections. For example, a collection could be the Car collection while a document could be the data item of a single car. Making the analogy with RDBMS, collections are similar to tables. Documents are similar to rows. Fields are similar to columns. MongoDB allows configuring a replica set. A replica set has primary replica set members and secondary replica set members (see Figure 3.2). There are two configurations based on the desired consistency level:

- Strong consistency. Applications write and read from the primary replica set member. The primary member will write all the operations that made it transact to the new

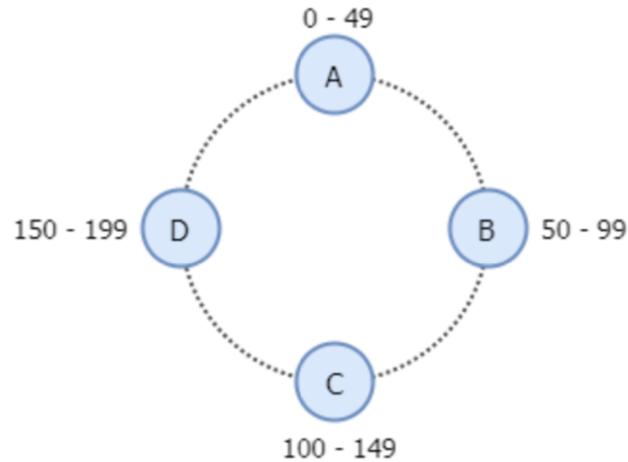


Figure 3.1: Cassandra ring [5].

state. These operations are idempotent and constitute the oplog (operations log). After the primary member acknowledges the application of the committed data and operations logging, secondary replica set members can now read from this log and replay all operations so that they can be on the same state of the primary member.

- Eventual consistency. Applications can read from secondary replica set members if they do not prioritize reading the latest data.

Oplog has a configurable back-limit history (default: 5% of the available disk space). If a secondary member fails longer enough to need operations that are no longer available in the oplog, all the databases, collections, and indexes directives are copied from the primary member or another secondary member. This process is called initial synchronization. The same one that is used when adding a new member to the replica set [5].

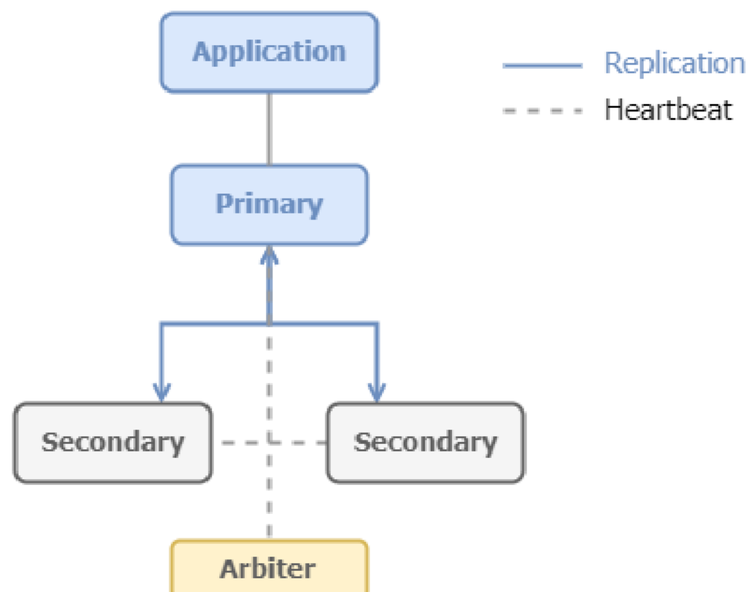


Figure 3.2: MongoDB replica set example [5].

The author compiles the whole comparison between Cassandra and MongoDB into a single

table (see Table 3.3) at the end of the article and concludes that MongoDB offers a more versatile approach, querying, and ease of use than Cassandra. The author makes a valuable theoretical analysis. However, it is not presented an experimental evaluation comparing Cassandra and MongoDB databases.

	Cassandra	MongoDB
Written in	Java	C++
Development model	Open-source	Open-source
Release year	2008	2009
NoSQL type	Column-oriented	Document
Data storage model	A keyspace consisting of column-families, and column-families consisting of rows.	Databases consisting of collections, and collections containing documents.
Schema	Flexible schema	No schema
Replication	Peer-to-peer	Master-slave
Sharding	Each server is a shard with its data replicated across other servers in the ring.	Each shard is a replica set. Thus, replicated data is stored on the same shard.
Consistency model	Eventual consistency	Strong consistency
CAP classification	AP	CP
Query language	CQL (similar to SQL)	JS-like syntax
Query model	Differs slightly compared to the data model (abstraction layer)	Identical to data model
Query abilities	Supports predictable queries, and are restricted to row keys.	Rich structure, supports many indexing options and dynamic queries.
Supporting resources	Official documentation is lacking, but has a large community and support.	Comprehensive documentation, community, and support.
Security features	Authentication, authorization, and encryption.	Authentication, authorization, auditing, and encryption.
Language and platform compatibility	There are currently 13 community-driven language drivers available, and Cassandra distributions only support Linux and macOS.	MongoDB officially offers 11 language drivers, and is available for all major platforms (Linux, macOS and Windows).

Figure 3.3: Cassandra vs MongoDB [5].

3.2 Analytical Models to Study Consistency on NoSQL Databases

Bailis et al. [26] suggest an approach that predicts the expected consistency of an eventually consistent Dynamo data store using models the authors developed called Probabilistically Bounded Staleness (PBS). This approach lacks a benchmark framework.

To analyze how eventual consistency is affected by the write and read consistency configurations offered by Cassandra, UC Berkeley developed a simulator called Probabilistically Bounded Staleness (PBS) [25]. Figures 3.4–8, show the resulting curves of a simulation using PBS, given the number of available cluster hosts (N), the read quorum (R) and the write quorum (W). They represent the probability of a client request having the latest version of the data over time (ms) for a given N, W and R combination. All the above configurations assume a $ReplicationFactor > 1$. If the Replication Factor were 1, there would be a single node storing a given data object, therefore the write operation and read operation would only execute in that single node resulting in strong consistency (as seen in Figure 3.4) for all configurations in Figures 3.4–8.

3.2.1 ALL Write Consistency Level or ALL Read Consistency Level

From Figure 3.4, we can conclude that the probability of consistency over time is constant, resulting in 100%. That is because each write or read operation is executed on every node available before acknowledging the result to the client. Therefore, this configuration makes the Cassandra cluster strong consistent.

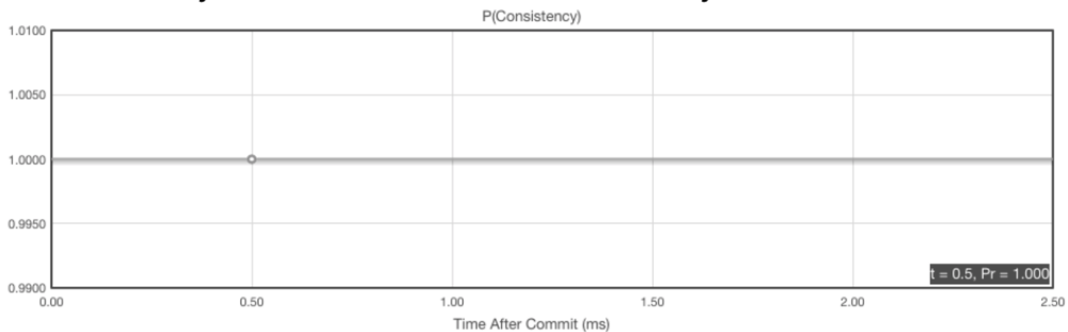


Figure 3.4: PBS results for (N = 5, R = 1, W = N = 5) and (N = 5, R = N = 5, W = 1).

3.2.2 ONE Read Consistency Level and QUORUM Write Consistency Level

In Figure 3.5, the consistency of a given data object eventually gets to 100%. A write operation needs three updated copies to acknowledge a successful write operation and a read operation returns the first copy the coordinator finds. The time that it is needed to reach 100% consistency is the time that the cluster needs to make all the number of copies previously set on the Replication Factor. With Read Consistency Level *ONE*, Cassandra will depend on the periodically Read Repair routines set by the Read Repair Chance to update all the copies of the data object and return all the time the same latest version.

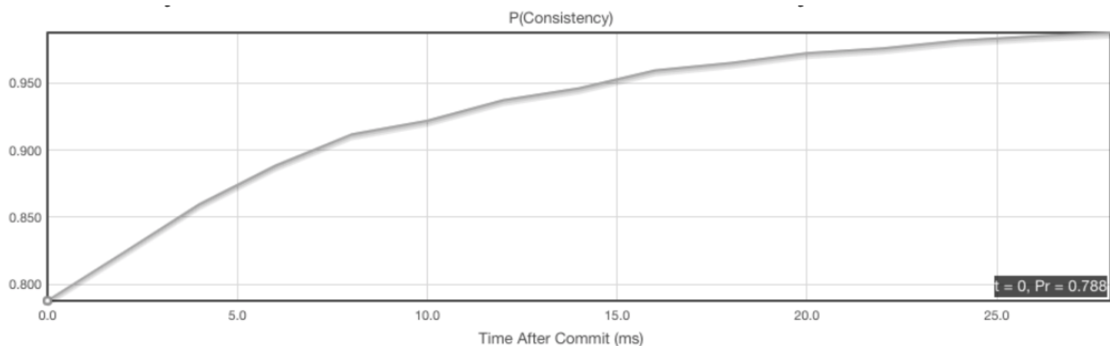


Figure 3.5: PBS results for (N=5, R=1, W=3).

3.2.3 QUORUM Read Consistency Level and ONE Write Consistency Level

From Figure 3.6, we can conclude that the time needed to reach full consistency of a given data object is the shortest of all configurations here (excluding the Figure 3.4 configuration). Three nodes are approached by the coordinator and the most updated version

among them is returned. For each read operation, Cassandra cluster uses its Read Repair feature to propagate to all three nodes inside the Quorum (the three nodes), so that they all have the most updated version of the requested data among them. Because Read Repair is always triggered by a read, the cluster reaches full consistency faster on the given data object.

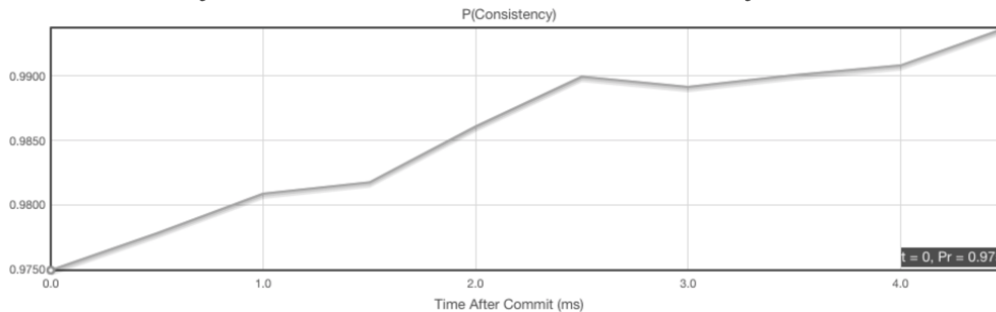


Figure 3.6: PBS results for (N=5, R=3, W=1).

3.2.4 ONE Read Consistency Level and ONE Write Consistency Level

In Figure 3.7, we have the strongest form of eventual consistency configuration in Cassandra. We need just one node with the updated data to acknowledge the write operation. For the reads, the first node the coordinator node chooses will retrieve the requested data. This may or may not be the most updated version of the data object. Eventually, the most updated version will be returned on all requests. The time needed to get to a 100% probability of consistency will depend on the Read Repair Chance and the Replication Factor. The higher the probability of the Read Repair Chance, the shorter the time to get to full consistency. The lower the Replication Factor, the shorter the time to get to full consistency. Modifying the Read Repair Chance and the Replication Factor to reach consistency faster will result in higher latencies because more copies and nodes are involved in the read and write operations for each client request.

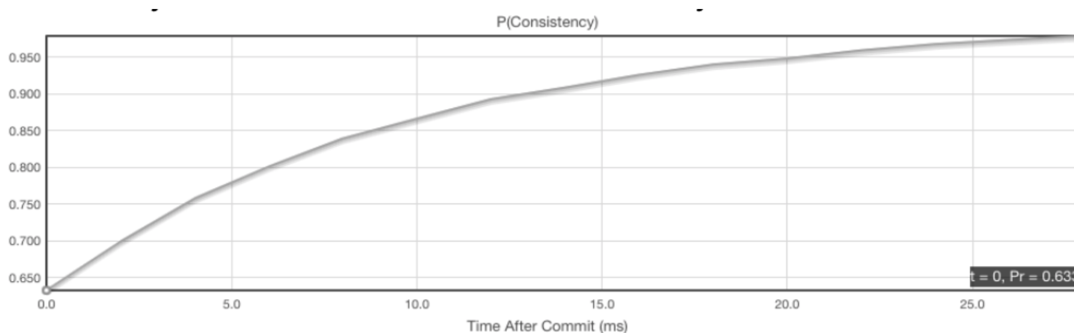


Figure 3.7: PBS results for (N=5, R=1, W=1).

3.3 Benchmark Consistency in NoSQL databases

This section presents the related work on building a consistency framework for NoSQL databases.

Cooper et al. [27] presents the YCSB (Yahoo! Cloud Serving Benchmark), a benchmarking framework for cloud serving systems. It comes to fill the need for performance comparisons between NoSQL databases and keep tracking of their tradeoffs such as read performance versus write performance, latency versus durability, and synchronous versus asynchronous replication. Although benchmark tiers such as performance and scaling are included in YCSB, it lacks other tiers such as availability and consistency. The tiers supported by YCSB are the following:

- Tier 1 - Performance. Performance tier focuses on the latency of requests when the database is under load.
- Tier 2 - Scaling. The Scaling tier of the database studies the impact on performance as more instances are added to the system.

Tudorica and Bucur present a critical comparison between NoSQL systems using multiple criteria [28]. The authors start to introduce multiple taxonomies to classify many NoSQL databases groups, even though there is not an official classification system on this type of databases. They define the following criteria to be used on the theoretical comparison: Persistence, replication, high availability, transactions, rack-locality awareness, implementation, influencers/sponsors, and license type. Tudorica and Bucur concentrate this theoretical comparison into one single table. Afterward, the authors make an empirical performance comparison, between Cassandra, HBase, Sherpa, and MySQL, using YCSB [11]. This article lacks other empirical metrics besides performance, such as consistency.

Patil et al. [29] propose a benchmark architecture that evaluates time to consistency. The authors extend the YCSB framework and add support to distributed architectures by using ZooKeeper for coordination. However, since 2011, development of the YCSB++ framework has been discontinued, and Cassandra support is still in progress. Only HBase and Acumulo support are available, but they are outdated as major releases of both databases have been released. YCSB++ also does not fully evaluate consistency trade-offs based on the CAP theorem as YCSB++ does not support network partition events.

Wang et al., in Reference [30], present a benchmarking effort on the replication and consistency strategies used in two databases: HBase and Cassandra. Wang et al. motivation are to evaluate tradeoffs, such as latency, consistency, and data replication. The authors conclude that in the latency of read/write operations is hardly improved by adding more replicas to the database. Higher levels of consistency dramatically increase write latency and are not suitable for reading the latest version of data and heavy writes in Cassandra. This paper lacks a more in-depth comparison of how consistency is affected in different configurations. Instead, this work is more focused on studying how consistency levels influence other properties in HBase and Cassandra databases.

The work of Bermbach and Tai [13] propose a benchmark methodology on Amazon's cloud database AWS S3. This is the closest work of what we aim to do in this thesis. Bermbach and Tai project a long-term monitor system on AWS S3 to evaluate how this service changes its consistency ability over time and the benchmark approach used can be easily extended for other usages and databases as we aim to demonstrate in this thesis. Bermback and Tai propose a benchmark methodology to study how Amazon S3 handles consistency over a long time period. This long-term experiment proposes a sin-gle writer and a variable number of readers as Figure 3.8 suggests. To achieve a uniformly load throughout the cluster's replicas and avoid always hitting the same replica, writer and readers interact with the cluster through a load balancer. During a benchmark the writer periodically persists a tuple (writeTimestamp;version) for each write operation and all readers record the tuple

(readTimestamp;writeTimestamp;version) for each read operation. An offline analysis or an online analysis lagging slightly behind can then detect consistency anomalies based on the reader logs [13].

Table 3.1 summarizes all the previous related work an how they are different from what we are proposing in this thesis.

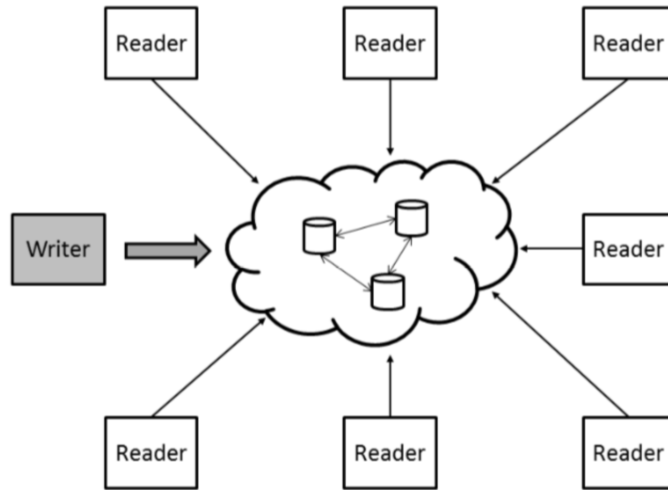


Figure 3.8: Bermback and Tai's propose benchmark architecture.

References	Method	Databases involved	Comparison with thesis
Bhamra [5]	Theoretical Comparison	Cassandra MongoDB	Lacks an empirical comparison
Bailis et al. [25]	Analytical Model	Dynamo databases	Lacks a benchmark framework
Cooper et al. [27]	Performance and Scaling Benchmark Framework (YCSB)	NoSQL databases	Lacks support for consistency, availability, and Network Partition Tolerance
Tudorica and Bucur [28]	Empirical comparison using YCSB	Cassandra HBase Sherpa MySQL	Lacks consistency measurements.
Patil et al. [29]	Distributed Benchmark Framework	NoSQL databases	No longer supports current versions of the proposed databases
Wang et al. [30]	Consistency Benchmark	HBase Cassandra	Lacks network partition support.
Bermbach and Tai [13]	Long-term consistency benchmark	Amazon S3	Lacks network partition support for non-cloud databases.

Table 3.1: Related work summary.

Chapter 4

CBench-Dynamo

A benchmark is a standardized tool to evaluate and compare competing systems or components according to specific characteristics. These characteristics can be performance, dependability, among others.

According to [12] the benchmarks can be categorized into three types: specification-based benchmarks, kit-based benchmarks, and a hybrid based on the latest two. Specification-based benchmarks are simulated based on a specific business problem by imposing certain functions that must be achieved, such as required input parameters and expected outcomes. This type of benchmark imposes a big development investment on presenting multiple implementations for the same problem and proceed with an evaluation of that set of development. While for specification-based benchmark, the specification is a set of rules implemented by the third party to load and run the benchmark. The Kit-based benchmarks use the specification as a guide for implementing the benchmark kit. A hybrid category can be provided mostly as a kit but allows some functions to be implemented depending on each individual benchmark run.

In this section, we propose CBench-Dynamo, a consistency benchmark that is a standard procedure to evaluate and compare consistency in the System Under Test (SUT). The specification to present proposes a benchmark approach to test consistency and availability in Dynamo-based NoSQL databases while subjecting these systems to network partition events. Therefore, this thesis contributes towards standardizing consistency benchmarking and lead vendors and buyers to better understand which system better suits their requirements.

4.1 CBench-Dynamo Properties

Benchmark researching and industry participants describe a benchmark into the following properties [12]: Relevance, Reproducibility, Fairness, Verifiability, and Usability.

Although the proposed benchmark, CBench-Dynamo, can be adapted to run in dedicated instances it has only been tested with Amazon EC2 instances. Some orchestration playbooks, such as easy instance setup, must be adapted to work on dedicated machines. However, as future work it is intended to make the framework more generic and versatile.

4.1.1 Relevance

Relevance is the most important property when defining a benchmark [12]. The relevance of a benchmark splits up into two dimensions, the spectrum of its applicability and the degree of relevance in the given area. The CBench-Dynamo is designed to target all Dynamo-based databases and the area of relevance is the study of the properties consistency, availability and network partition tolerance, of a horizontal-distributed database system. This benchmark aims to be a framework to facilitate the decision process of choosing the most appropriate NoSQL database depending on the degree of performance, availability, consistency, and network fault tolerance required for running a given application.

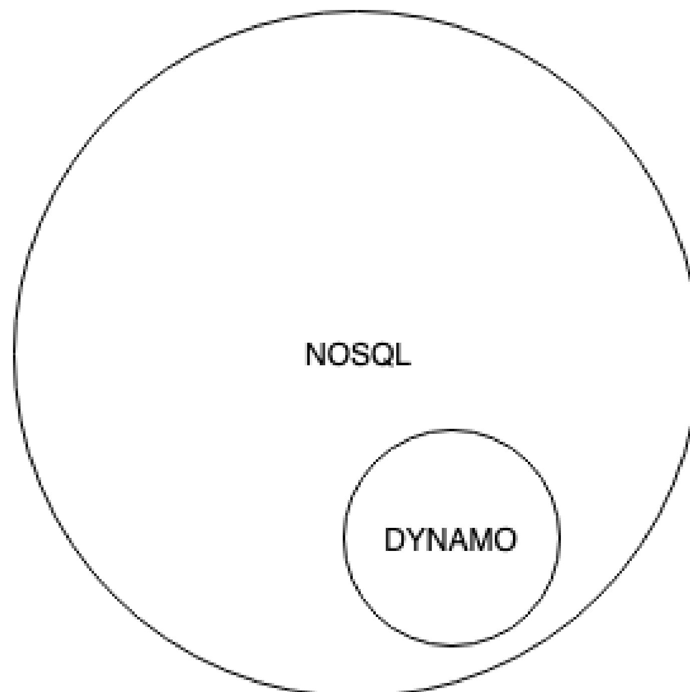


Figure 4.1: Dynamo-based databases set within NoSQL realm.

4.1.2 Reproducibility

Reproducibility will be attained as CBench-Dynamo exports the instances' hardware and software facts via Ansible. In addition, the workload specs will be also exported at the end of the associated run. The goal of this extensive and detailed description is for other people to obtain identical results by configuring the whole system as described.

All the proposed benchmark modules are hosted on GitHub. There is a repository for each of these modules, modified YCSB [14], analyzer [16], and orchestration playbooks [15].

```
{
    "ansible_all_ipv4_addresses": [
        "REDACTED IP ADDRESS"
    ],
    "ansible_all_ipv6_addresses": [
        "REDACTED IPV6 ADDRESS"
    ],
    "ansible_apparmor": {
        "status": "disabled"
    },
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "11/28/2013",
    "ansible_bios_version": "4.1.5",
    "ansible_cmdline": {
        "BOOT_IMAGE": "/boot/vmlinuz-3.10.0-862.14.4.el7.x86_64",
        "console": "ttyS0,115200",
        "no_timer_check": true,
        "nofb": true,
        "nomodeset": true,
        "ro": true,
        "root": "LABEL=cloudimg-rootfs",
        "vga": "normal"
    },
    "ansible_default_ipv4": {
        "address": "REDACTED",
        "alias": "eth0",
        "broadcast": "REDACTED",
        "gateway": "REDACTED",
        "interface": "eth0",
        "macaddress": "REDACTED",
        "mtu": 1500,
        "netmask": "255.255.255.0",
        "network": "REDACTED",
        "type": "ether"
    }
    (...)
}
```


Figure 4.2: Example of Ansible facts. [6]

4.1.3 Fairness

Fairness is the ability of the results being supported by the system merit without artificial constraints. To reach fairness a set of artificial constraints must be consent and well defined. CBench-Dynamo defines the following constraints:

- The SUT must be a Dynamo-based NoSQL database system, e.g. Cassandra;
- The SUT must have the same hardware, network and operating system components when comparing benchmark test results targeting similar SUT;

- The Workload Coordinator must support a JVM to run the benchmark test and Python to analyze and translate the data into meaningful measurements;

- The fact that the Workload Coordinator uses Java and Python to coordinate the benchmark and post-process all the data, respectively, makes the system highly portable and therefore fair as JVM and Python based applications can run virtually in any system.

4.1.4 Verifiability

It is important that results are trustworthy. Results must be validated and decrease the possibility of chance or manipulation. CBench-Dynamo results from academic work, all the workloads presented here were subject to peer-review by other researchers.

4.1.5 Usability

Usability is the degree of how easy a system is to use. Several layers of abstractions were taken into account so that only a minimum input is needed to start a benchmark test.

4.2 Methodology

This paper proposes a workload that evaluates how consistency, performance, and availability are affected when consistency is configured either to prefer a high-available system or a high-consistent system while in a distributed system, such as Dynamo-based databases, where network partition events may occur. The proposed workload is a customized YCSB workload and follows the method-ology proposed by Bermback and Tai [13]. Bermback and Tai propose a benchmark methodology to study how Amazon S3 handles consistency over a long time period. This long-term experiment proposes a single writer and a variable number of readers. To achieve a uniformly load throughout the cluster's replicas and avoid always hitting the same replica, writer and readers interact with the cluster through a load balancer.

Our benchmark is composed of two stages, the load, and run stages. The load stage is off the record for benchmark purposes. This stage's goal is to load all the objects into the database. These objects are composed only of two fields, key, and version. During the benchmark run phase, both update and read operations occur uniformly. When configuring a workload run, the parameters threads indicate how much writers and readers will be running, as only one writer is used, the number of readers is calculated as threads-1. Each write operation increments a given object's version and each read operation reads the version of a given object. The writer and the readers each have their task plan pre-generated at the beginning of the test and the benchmark ends when all the objects have been updated and read from until the pre-configured final version. Each operation is registered into a common file (see Figure 4.3).

```
(...)
writer_id:0, key:9345f1bae61442dab3f167c02d19a4a8,
timestamp:17309459474301, version:1
(...)
reader_id:2, key:9345f1bae61442dab3f167c02d19a4a8,
timestamp:17309253174394, version:0
reader_id:1, key:9345f1bae61442dab3f167c02d19a4a8,
timestamp:17309253174398, version:UNAVAILABLE
(...)
```

Figure 4.3: Proposed benchmark's results data structure.

The generated data is sufficient to infer whether a consistency anomaly had happened. For a given object's key, if a read operation returns a version inferior to a version already written by some write operation in the past, there was a consistency anomaly. At the same time this occurs, there is a module that is disconnecting from time to time one instance at a time from the cluster to simulate network partition events. For every operation that the cluster was not able to retrieve a successful answer, the version assumes the UNAVAILABLE value as 4.3 suggests. All consistency anomalies are then processed and translated into the following measurements: availability probability, consistency probability, write latency, and read latency.

4.3 Architecture Specification

CBench-Dynamo is composed of a Workload Coordinator, a Load Balancer and a Dynamo cluster (SUT). All these components are orchestrated by the orchestrator via Ansible playbooks, as illustrated in Figure 4.4.

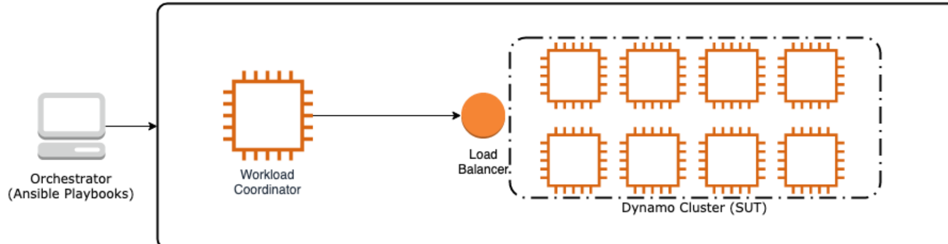


Figure 4.4: CBench-Dynamo Architecture Specification.

4.3.1 Orchestrator

As we are benchmarking a distributed system and with a easy replication in mind, there are many processes to manage and to sync up with one another. Hence, the Orchestrator is the module that handle all this management and synchronization between all the existing modules from benchmark preparation to benchmark final results.

The Orchestrator was developed as multiple Ansible playbooks. Ansible is an open source IT configuration management, deployment, and orchestration tool. Ansible enables clear orchestration of complex multi-tier workflows under a single point of management.

Name	Instance ID	Instance Type	Availability Zone
coordinator	i-0ec212a95302aba79	c5n.xlarge	eu-west-1a
node	i-00c3cb8ae9b5b2f6e	m5d.large	eu-west-1c
node	i-0348fc8ebfd63da6a	m5d.large	eu-west-1c
node	i-07ef2ee3c70d10d24	m5d.large	eu-west-1c
node	i-0960e6433e7bbbd0c	m5d.large	eu-west-1c
node	i-0c2acea700ed9d42e	m5d.large	eu-west-1c
node	i-0c6f2b05fc885edbd	m5d.large	eu-west-1c
node	i-0f29704eb7ae64978	m5d.large	eu-west-1c
node	i-0ffb368e29edf2d86	m5d.large	eu-west-1c

Figure 4.5: Cassandra nodes and coordinator instances listed on AWS Management Console.

Ansible was used for automating the following tasks:

1. setup the SUT
2. setup de Workload Coordinator
3. prepare the SUT for testing
4. run benchmark test
5. clean SUT
6. trigger the analyser against the generated data

An Ansible project is composed by playbooks and roles. A playbook is a script file composed by tasks targeting the remote host defined in *hosts*. Figure 4.6 represents an example of a playbook. The hosts can be a group of IP addresses as long as they are defined in the file */etc/ansible/hosts* (see Figure 4.7), or a IP address or host name. A role is much alike a playbook. It is a way to better organize an Ansible project as roles can be run by more than one playbook. Figure 4.8 illustrates a generic role.

```
---
- hosts: nodes
    remote_user: ubuntu
    gather_facts: yes
    roles:
        - run_this_role.yml
- hosts: Add local hostname to /etc/hosts
remote_user: ubuntu
gather_facts: yes
roles:
    - run_this_second_role.yml
```

Figure 4.6: Ansible generic playbook.

```
[nodes]
node01 ansible_host=34.244.127.84
node02 ansible_host=54.72.103.106
node03 ansible_host=34.244.237.236
node04 ansible_host=18.202.36.92
node05 ansible_host=52.49.176.148
node06 ansible_host=18.202.253.1
node07 ansible_host=34.253.9.164
node08 ansible_host=52.48.5.184

[coordinator]
co ansible_host=52.213.81.171

[load-balancer]
lb-1108247867.eu-west-1.elb.amazonaws.com

[local]
127.0.0.1
```

Figure 4.7: Ansible hosts file.

```
# file: run_this_role.yml
---
- name: Description of the task here.
    module:
    arg1: value
    arg2: value
    arg3: value
    ....
    become: yes|no
```

Figure 4.8: Ansible generic role.

The first three dashes in playbooks and roles indicate that we are in presence of a YAML file. Ansible language of choice is YAML.

A task is composed by a **name**, a **module** call, and, besides other arguments, an argument called **become**.

name. The name is a text that describes what we intend to do with the current task.

module. The module is some pre-made program that receives a variable number of arguments. The modules run on the remote host.

become. The argument *become* allow us to run the module with administrator privileges (*sudo*) on the remote host. This feature is enable when the value passed is *yes*. The default value is *no*.

In order to group the remote machines we want to target with our playbooks we need to edit the file */etc/ansible/hosts* (see Figure 4.7) with the IP addresses provided by AWS EC2 instances.

Setting up Cassandra

The code snippets in Figure 4.9 and Figure 4.3.1 represent the playbook and the role respectively responsible for setting up Cassandra cluster. Here we assume that we are handling a clean EC2 machine from Amazon AWS and from scratch we configure a Cassandra cluster based on a cluster configuration followed by previous configured machines so that they can join the same Cassandra cluster. The setting up process goes from downloading and installing the required dependencies to

finally run Cassandra service.

```
# file: setup.yml
---
 - hosts: nodes
   remote_user: ubuntu
   gather_facts: yes
   become: yes
   roles:
      - setup_cassandra
 - hosts: local
   connection: local
   roles:
      - prepare_cassandra
```

Figure 4.9: Ansible playbook for setting up and preparing for testing a Cassandra node in a AWS EC2 instance.

```
# file: setup_cassandra.yml
---
- name: Add local hostname to /etc/hosts
    lineinfile:
    dest: /etc/hosts
    line: "127.0.0.1 {{ansible_hostname}}"
    state: present
    become: yes
- name: Add Cassandra repo to apt
    apt_repository:
    repo: deb http://www.apache.org/dist/cassandra/debian 311x main
    filename: cassandra.sources.list
    state: present
    update_cache: no
- name: Update apt
    apt:
    update_cache: yes
    upgrade: yes
    allow-unauthenticated: yes
- name: install openjdk-8-jre
    apt:
    name: openjdk-8-jre
- name: install python
    apt:
    name: python
```

```
- name: Install Cassandra
    apt:
    name: cassandra
    allow-unauthenticated: yes
- name: Stop Cassandra service
    service:
    name: cassandra
    state: stopped
- name: Remove all data files
    shell: rm -rf /var/lib/cassandra/data/*
    become: yes
- name: Override cassandra.yaml config file
    template: src=cassandra.yaml dest=/etc/cassandra/
- name: run Cassandra service
    service:
    name: cassandra
    state: started
    become: yes
```

Figure 4.10: Ansible role for setting up a Cassandra node in a AWS EC2 instance.

Preparing Cassandra for testing

The code snippet in 4.11 represents the role for preparing a single Cassandra node. Here we have two tasks targeting a single node that is responsible for propagating this configuration through out all the nodes from the cluster. The first task creates the namespace *ycsb* with *replication factor* of 3, and the second one creates the table where the workload data is written.

```
# file: prepare_cassandra.yml
---
- name: Create Cassandra Keyspace
    shell: "cqlsh {{load_balancer_ip_address}} -e \"CREATE KEYSPACE IF
        NOT EXISTS ycsb WITH REPLICATION = { 'class' : 'SimpleStrategy
        ', 'replication_factor' : 3 };\""
- name: Create Workload Table
    shell: "cqlsh {{load_balancer_ip_address}} -e \"USE ycsb; CREATE
        TABLE IF NOT EXISTS workload (y_id varchar primary key, version
         varchar);\""
```

Figure 4.11: Ansible playbook for preparing a Cassandra node in a AWS EC2 instance.

Setting up the Workload Coordinator

The workload coordinator is responsible for running CBench-Dynamo workloads. The following code snippets (Figure 4.12 and Figure 4.13) are the playbook and the role respectively for setting up the coordinator.

```
# file: coordinator.yml
---
- hosts: coordinator
  remote_user: ubuntu
  gather_facts: yes
  become: yes
  roles:
    - setup_coordinator
```

Figure 4.12: Ansible playbook for setting up the benchmark coordinator.

```
# file: setup_coordinator.yml
---
- name: Add local hostname to /etc/hosts
  lineinfile:
    dest: /etc/hosts
    line: "127.0.0.1 {{ansible_hostname}}"
    state: present
  become: yes
- name: Update apt
  apt:
    update_cache: yes
    upgrade: yes
    allow-unauthenticated: yes
- name: install openjdk-8-jdk
  apt:
    name: openjdk-8-jre
- name: install maven
  apt:
    name: maven
- name: install python
  apt:
    name: python
```

Figure 4.13: Ansible role for setting up the benchmark coordinator.

Running the benchmark

The following code snippet 4.14 represents de playbook that runs the benchmark. This playbook calls the role *run_benchmark.yml* (see Figure 4.15) that will be called for each map of variables defined bellow the field *with_items*. All the arguments defined bellow *vars* will be common to all benchmark iterations. In summary, we will have 5 workloads and they are represented in Table 4.1.

#	Description	Driver	Write consistency	Read consistency	Threads	Objects	Versions
1	cassandra	cassandra -cql	*ALL*	ALL	4	1000000	2
2	cassandra	cassandra -cql	*ONE*	ONE	4	1000000	2
3	cassandra	cassandra -cql	*ONE*	QUORUM	4	1000000	2
4	cassandra	cassandra -cql	*QUORUM*	ONE	4	1000000	2
5	cassandra	cassandra -cql	*QUORUM*	QUORUM	4	1000000	2

Table 4.1: Workload configurations.

```
# file: benchmark.yml
---
- hosts: coordinator
  remote_user: ubuntu
  tasks:
  - name: Use role in loop
    include_role:
      name: run_benchmark
    vars:
      - description: "{{ item.description }}"
      - db: "{{ item.db }}"
      - write_consistency: "{{ item.write_consistency }}"
      - read_consistency: "{{ item.read_consistency }}"
      - threads: "{{ item.threads }}"
      - objects_num: "{{ item.objects_num }}"
      - version_num: "{{ item.version_num }}"
      - host: "{{ groups['nodes'] | random }}"
    with_items:
        - { description: cassandra, db: cassandra-cql,
            write_consistency: ALL, read_consistency: ALL, threads: 4,
            objects_num: 1000000, version_num: 2 }
        - { description: cassandra, db: cassandra-cql,
            write_consistency: ONE, read_consistency: ONE, threads: 4,
            objects_num: 1000000, version_num: 2 }
        - { description: cassandra, db: cassandra-cql,
            write_consistency: ONE, read_consistency: QUORUM, threads:
            4, objects_num: 1000000, version_num: 2 }
        - { description: cassandra, db: cassandra-cql,
            write_consistency: QUORUM, read_consistency: ONE, threads:
            4, objects_num: 1000000, version_num: 2 }
        - { description: cassandra, db: cassandra-cql,
            write_consistency: QUORUM, read_consistency: QUORUM,
            threads: 4, objects_num: 1000000, version_num: 2 }
```

Figure 4.14: Ansible playbook for running a set of benchmark configurations.

```
# file: run_benchmark.yml
---
- name: Reboot AWS instances
  local_action:
    module: ec2
    aws_access_key: "{{aws_access_key}}"
    aws_secret_key: "{{aws_secret_key}}"
    region: eu-west-1
    wait: yes
    instance_tags:
        Name: node
    state: restarted

- name: With AWS instances started
  local_action:
    module: ec2
    aws_access_key: "{{aws_access_key}}"
    aws_secret_key: "{{aws_secret_key}}"
    region: eu-west-1
    wait: yes
    instance_tags:
        Name: node
    state: running

- name: sleep for 60 seconds and continue with play
  wait_for:
    timeout: 60

- name: "Load {{description}} write_consistency={{write_consistency}},
    read_consistency={{read_consistency}}"
  shell: "chdir=~/ycsb ./bin/ycsb load {{db}} -p hosts={{
      load_balancer_ip_address}} -p threadcount={{threads}} -p
      objectversionlimit={{version_num}} -p numobjects={{objects_num}}
      -p cassandra.readconsistencylevel={{read_consistency}} -p
      cassandra.writeconsistencylevel={{write_consistency}} -P
      workloads/myworkload -s > ~/load.log"

- name: "Run {{description}} write_consistency={{write_consistency}},
    read_consistency={{read_consistency}}"
  shell: "chdir=~/ycsb ./bin/ycsb run {{db}} -p hosts={{
      load_balancer_ip_address}} -p threadcount={{threads}} -p
      objectversionlimit={{version_num}} -p numobjects={{objects_num}}
      -p cassandra.readconsistencylevel={{read_consistency}} -p
      cassandra.writeconsistencylevel={{write_consistency}} -P
      workloads/myworkload -s > ~/reports/ycsb/{{ansible_date_time.
      epoch}}_{{description}}_w{{write_consistency}}_r{{
      read_consistency}}.dat"
```

Figure 4.15: Ansible role for running a single benchmark configuration.

4.3.2 Load Balancer

The load balancer is a basic AWS Network Load Balancer. Its purpose is to balance the database calls coming from the benchmark testing evenly across the 8 nodes of the cluster. Figure 4.16 illustrates the load balancer instance used on AWS Management Console.

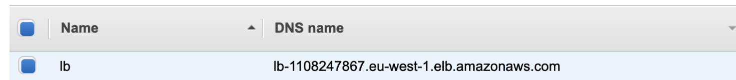


Figure 4.16: Load balancer instance used listed on AWS Management Console.

4.3.3 Data Analyser

The workload coordinator is responsible for running CBench-Dynamo workloads. The role in Figure 4.17 runs the python script (see code snippet in Figure 4.18) responsible for measuring the data produced by the workload. The following measurements are calculated:

- Availability probability
- Consistency probability
- Latency (write)
- Latency (read)

```
---
- name: "Analyse {{filename}}"
  shell: "chdir=~/anayser python -m bin.main run ~/reports/ycsb/ ~/
      reports/analyser//{{ansible_date_time.epoch}}_analysis.csv"
  register: out

- debug: msg="{{ out.stdout }}"
```

Figure 4.17: Ansible role for analysing data.

```
def calculate(self):
    inconsistencies_reads_counter = 0
    total_reads = 0
    for entry in self.data:
        if entry.worker_type == DataEntry.WRITER:
            self.registry[entry.key] = (entry.timestamp, entry.version)
        else:
            total_reads = total_reads + 1
            timestamp, version = self.registry.get(entry.key, (0, 0))
            if int(entry.version) < int(version) and int(timestamp) <
                int(entry.timestamp):
                 inconsistencies_reads_counter =
                     inconsistencies_reads_counter + 1
                 print(entry)

    consistent_reads_counter = total_reads -
       inconsistencies_reads_counter

    try:
        self.ratio_read_latency_and_consistency_score = self.
            read_average_latency / (
                 consistent_reads_counter / total_reads)
    except ZeroDivisionError:
        self.ratio_read_latency_and_consistency_score = 0
    try:
        self.ratio_write_latency_and_consistency_score = self.
            write_average_latency / (
                 consistent_reads_counter / total_reads)
    except ZeroDivisionError:
        self.ratio_write_latency_and_consistency_score = 0

    if consistent_reads_counter == 0:
        self.consistency_score = 0
    else:
        self.consistency_score = consistent_reads_counter / total_reads
             * 100

    if self.unavailable_service_counter == self.total_operations:
        self.availability_score = 0
    else:
        self.availability_score = (self.total_operations - self.
            unavailable_service_counter) / self.total_operations * 100
```

Figure 4.18: Main logic to calculate the measurements evaluated.

4.3.4 Network partition events generator

The network partition events generator's playbook (Figure 4.19) and role (Figure 4.20) are responsible for calling the bash script that injects the fault (Figure 4.21) for each cluster node. This node is randomly chosen and it stops operating, hence simulating a network partition event. The script stops the Cassandra node represented by the IP address it receives from the arguments from the playbook call and after stopping the node the script sleeps for a specific amount of time and then restarts the node it just stopped. This repeats until the benchmark is over.

```
---
- hosts: local
  connection: local
  tasks:
  - name: Use role in loop
    include_role:
      name: inject_faults
    vars:
      - service: "{{ item.service }}"
      - fault_duration: "{{ item.fault_duration }}"
      - fault_interval: "{{ item.fault_interval }}"
    with_items:
        - { service: cassandra, fault_duration: 2, fault_interval:
          25 }
```

Figure 4.19: Ansible playbook for initiating the network partition event generator.

```
---
- name: With AWS instances started
    local_action:
    module: ec2
    aws_access_key: "{{aws_access_key}}"
    aws_secret_key: "{{aws_secret_key}}"
    region: eu-west-1
    wait: yes
    instance_tags:
        Name: node
    state: running

- name: gather facts from nodes
    setup:
    delegate_to: "{{facts_item}}"
    delegate_facts: True
    remote_user: ubuntu
    loop: "{{groups['nodes']}}"
    loop_control:
    loop_var: facts_item

- name: Interpolate fault_injection.sh
    template: src=fault_injection.sh dest=fault_injection.sh

- name: "Injecting network partition events in the background (see
   script.log)"
    shell: "sh fault_injection.sh {{fault_duration}} {{fault_interval}}
        {{service}} ubuntu > script.log"
```

Figure 4.20: Ansible role that randomly chooses the cluster node to fail and calls the network partition event bash script.

```
while true
do
    RANDOM=$$$(date +%s)
    IP=${IPS[$RANDOM % ${#IPS[@]}]}
    echo initiating service fault on ${IP}
    ssh ${USER}@${IP} 'sudo service '${SERVICE}' stop'
    echo service ${SERVICE} stopped on ${IP}
    RANDOM_FAULT_DURATION=$(( ( RANDOM % ${FAULT_DURATION_MAX} ) + 1 )
        )
    echo sleeping ${RANDOM_FAULT_DURATION}s before starting ${SERVICE}
        again on ${IP}...
    sleep ${RANDOM_FAULT_DURATION}
    ssh ${USER}@${IP} 'sudo service '${SERVICE}' start'
    echo service ${SERVICE} started
    RANDOM_FAULT_INTERVAL=$(( ( RANDOM % ${FAULT_INTERVAL_MAX} ) + 1 )
        )
    echo sleeping ${RANDOM_FAULT_INTERVAL}s before next fault...
    sleep ${RANDOM_FAULT_INTERVAL}
    echo
done
```

Figure 4.21: Bash script that stops Cassandra service on the cluster node received as argument, sleeps for specific time, and recovers the cluster node back to life.

The output of the playbook will give details about the network partition events that are currently taking place. Figure 4.22 illustrates a possible output.

```
initiating service fault on 34.244.127.84
service cassandra stopped on 34.244.127.84
sleeping 1s before starting cassandra again on 34.244.127.84...
service cassandra started
sleeping 7s before next fault...

initiating service fault on 34.244.237.236
service cassandra stopped on 34.244.237.236
sleeping 1s before starting cassandra again on 34.244.237.236...
service cassandra started
sleeping 16s before next fault...

initiating service fault on 52.49.176.148
service cassandra stopped on 52.49.176.148
sleeping 2s before starting cassandra again on 52.49.176.148...
service cassandra started
sleeping 7s before next fault...

initiating service fault on 34.244.237.236
service cassandra stopped on 34.244.237.236
sleeping 2s before starting cassandra again on 34.244.237.236...
service cassandra started
sleeping 4s before next fault...
```

Figure 4.22: Network partition events generator output.

Chapter 5

Experimental Evaluation

As the first test of the proposed benchmark, Cassandra was the chosen SUT. Cassandra is a database system built with distributed systems in mind, like almost every NoSQL systems out there. Following the CAP theorem, Cassandra by default is on the AP (Availability and Network Partition Tolerance) side, hence prioritizing high-availability when subject to network partitioning. As we will further see, Cassandra's consistency can be tuned to be a CP (Consistency and Network Partition Tolerance) database system, so it becomes a strong consistent database when subject to network partitioning [17].

5.1 Cassandra

Cassandra is a column NoSQL database [31]. It was initially developed by Facebook to fulfill the needs of the company's Inbox Search services. In 2009, it became an Apache Project.

Cassandra is a database system built with distributed systems in mind, like almost every NoSQL systems out there. Following the CAP theorem, Cassandra will be on the AP (Availability and Network Partition Tolerance) side, hence prioritizing high-availability when subject to network partitioning. As we will further see, Cassandra's consistency can be tuned to be a CP (Consistency and Network Partition Tolerance) database system, so it becomes a strong consistency database when subject to network partitioning. Cassandra system is a column based NoSQL database [5]. In other words, Cassandra describes data by using columns. A keyspace is the outermost container for the entire dataset, corresponding to the entire database, and it is composed of many column-families. A column-family represents the same class of objects, like a Car or a Person, and each column-family has different entries of objects called rows. Each row is uniquely identified by a row key or partition key and can hold an arbitrarily large number of columns. A column contains a name-value pair and a timestamp. This timestamp is necessary when solving consistency conflicts.

Cassandra scales up by distributing data across a set of nodes, designated as a cluster. Each node is capable of answering client requests. When a node is working on a client request, it becomes the coordinator for that request. It will be responsible for asking to other nodes for the requested data and answering back to the application.

Cassandra partitions data across the cluster by hashing the row key. Each node on the ring stores a subset of hashes, in such a way that "the largest hash value wraps around the smallest hash value" [31]. Because of the randomness of the hash functions, data tends to be evenly distributed across the ring.

Replication is the strategy Cassandra uses to achieve a high-available system. Two concepts describe a replication configuration in Cassandra, replication strategy and replication factor [32]. The Replication strategy determines which nodes replicas are placed. The replication factor determines how many different nodes have the same data. If the replication factor is R, the node that is responsible for that specific key range copies the data it owns to the next $R-1$ neighbors,

clockwise as in Figure 2.4.

Cassandra was initially designed to be eventually consistent, high-available and low-latency. However, its consistency can be tuned to match the client's requirements. The following configuration constants describe some of the different write consistency levels [33]:

- ***ALL***. Data is written on all replica nodes in the cluster before the coordinator node acknowledges the client. (Strong Consistency, high latency)
- ***QUORUM***. Data is written on a given number of replica nodes in the cluster before the coordinator node acknowledges the client. This number is called the quorum. (Eventual Consistency, low latency)
- ***LOCAL_QUORUM***. Data is written on a quorum of replica nodes in the same data center as the coordinator node before this last one acknowledges the client. (Eventual Consistency, low latency)
- ***ONE***. Data is written in at least one replica node. (Eventual Consistency, low latency)
- ***LOCAL_ONE***. Data is written in at least one replica node in the same data center as the coordinator node. (Eventual Consistency, low latency)

5.2 Architecture

CBench-Dynamo requires an architecture composed of an orchestrator, a workload coordinator, and a dynamo cluster as the SUT. The following architecture was de-fined for our first test (see Figure 5.1):

Orchestrator. The orchestrator is a MacBook Pro 13-inch, 2017, 2.3GHz Intel Core i5 with 8GB of RAM.

Workload Coordinator. The workload coordinator is an Amazon EC2 C5n.xlarge instance (4 vCPUs, 10.5GB RAM).

SUT. The System Under Test is a Cassandra cluster composed of eight Amazon EC2 M5d.large instances. Each cluster instance will be rebooted and reloaded between workloads by the Orchestrator and the Workload Coordinator.

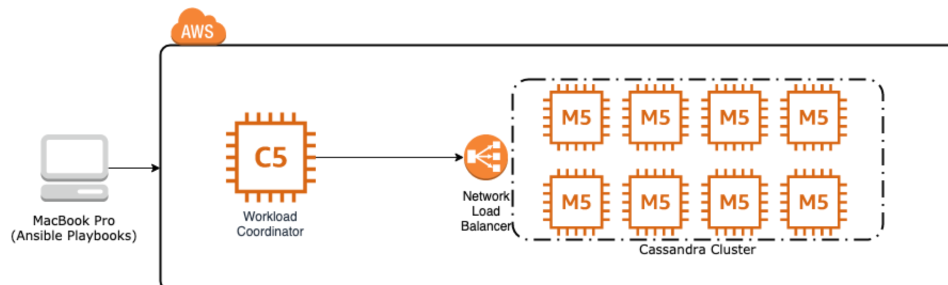


Figure 5.1: Testing architecture.

5.3 Experiment

In this section, we present the results obtained after running the proposed workload configurations. Our experiment targets an 8-node Cassandra cluster and combines different consistency configurations, i.e. *ONE*, *QUORUM*, *ALL*, as described in Table 5.1. The common input parameters for every configuration are the following:

- Replication Factor: 3;

- Total number of objects: 1.000.000;
- Versions/updates per object: 2;
- Network partition event duration: 2s;
- Interval between network partition events: random between [1s, 25s].

Configuration	SUT	Write Consistency	Read Consistency
1	Cassandra	*ALL*	*ALL*
2	Cassandra	*ONE*	*ONE*
3	Cassandra	*QUORUM*	*ONE*
4	Cassandra	*ONE*	*QUORUM*
5	Cassandra	*QUORUM*	*QUORUM*

Table 5.1: Cassandra's benchmark workload configurations.

Consistency and Availability. When in a configuration where both read and write consistency is *ALL* we achieve results of Strong Consistency while compromising availability. This happens as theorized because all replicas must be involved before returning to the client. If some replica is down, resulting from a network partition event, the request can't fulfill and the response to the client reports an unavailable service. Although this configuration generated an availability of 99.7539%, the industry does not consider this value high. Availability is usually represented by how many nines the availability probability has (see Table 5.2). The value we attained in the *ALL-ALL* configuration only has 2-nines, which means that this number only falls into the second level of availability, hence translating into 3.65 days of availability when rounding down the number to 99.0000%. Many businesses that require high-availability may fail with such a long unavailable service time.

In the other hand, when querying Cassandra with no consistency constraints by set-ting both read and write operations to involve just one replica (write consistency = *ONE* and read consistency = *ONE*), we achieved 100% of availability, but we have compromised consistency down to the lowest value achieved in the whole experience.

As of configurations using *QUORUM* combined with *ONE*, we achieved a more balanced consistency/availability relation. As we had chosen a replica factor of three, the *QUORUM* involves two replicas when processing a client request. When reading with *ONE* and writing with *QUORUM*, the request may involve the third replica that was not part of the *QUORUM* for that given data object, hence returning an outdated version. When inverting the order, the ONE in the writing and the *QUORUM* in the reading, it seems not to have such a drastic decline in consistency, however availability loses a nine.

For a *QUORUM-QUORUM* configuration, we achieved strong-consistency and high-availability. This configuration can tolerate some network partition events unless the number of replicas down compromises the quorum. Because network partition events had disconnected one replica at a time, the quorum had never been compromised, hence the results we had and represented in Figure 5.2.

Performance. The second analysis is in terms of read and write operation latencies given a consistency setting. As Figure 5.3 illustrates, for an *ALL-ALL* configuration we achieved as expected the

Availability (%)	Downtime per year
90.0000 (one nine availability)	36.53 days
99.0000 (two nines availability)	3.65 days
99.9000 (three nines availability)	8.77 hours
99.9900 (four nines availability)	52.60 minutes
99.9990 (five nines availability)	5.26 minutes
99.9999 (six nines availability)	31.56 seconds

Table 5.2: Availability and nines notation [7]

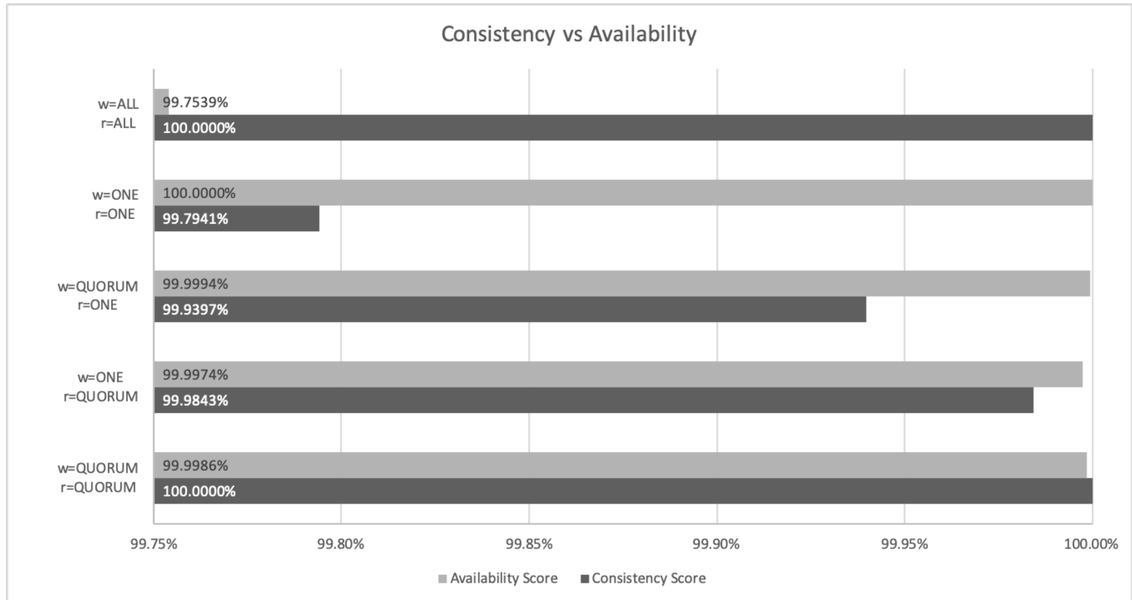


Figure 5.2: Consistency and availability results for all configurations.

highest latencies of all configurations because all replicas had to be involved in read and write operations. For the *ONE-ONE* configuration, because only one replica needed to be involved in read and write operations, the latencies are the lowest among all configurations tested when combining the two latencies. However, when compared solely on mixed *ONE-QUORUM* configurations, ONE latency in these last configurations are better. Finally, for *QUORUM-QUORUM* configuration we achieved the most balanced configuration between latency, consistency, and availability.

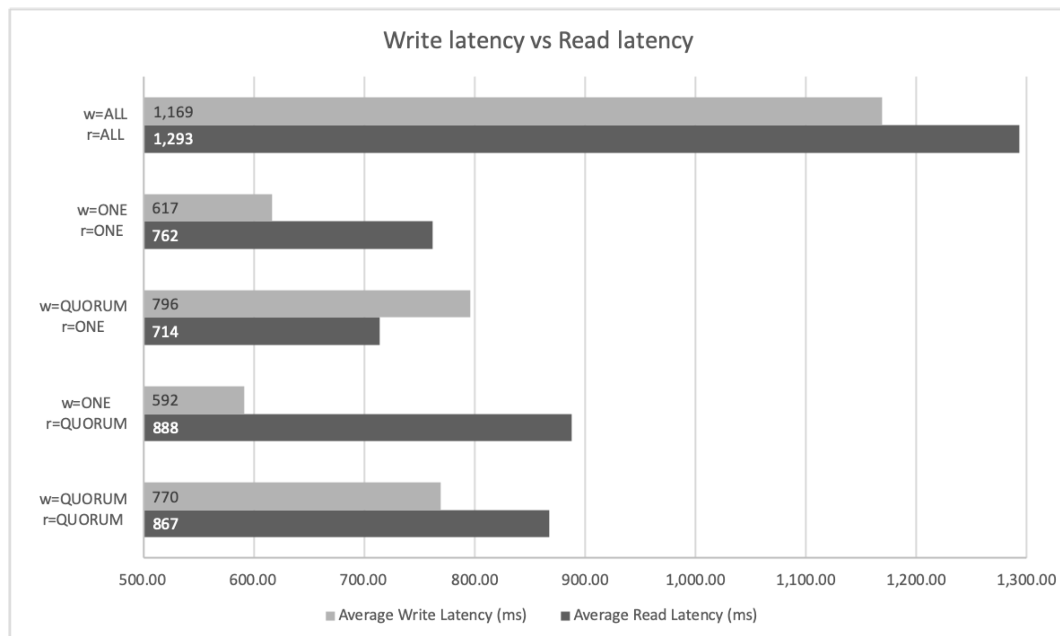


Figure 5.3: Write latency and read latency results for all configurations.

Chapter 6

Work Plan

This thesis work is structured into four main parts, the survey, the proof of concept, the implementation and analyses, and conclusions. A more detailed representation of the work planning can be found in the Gantt diagram located in Appendix A.

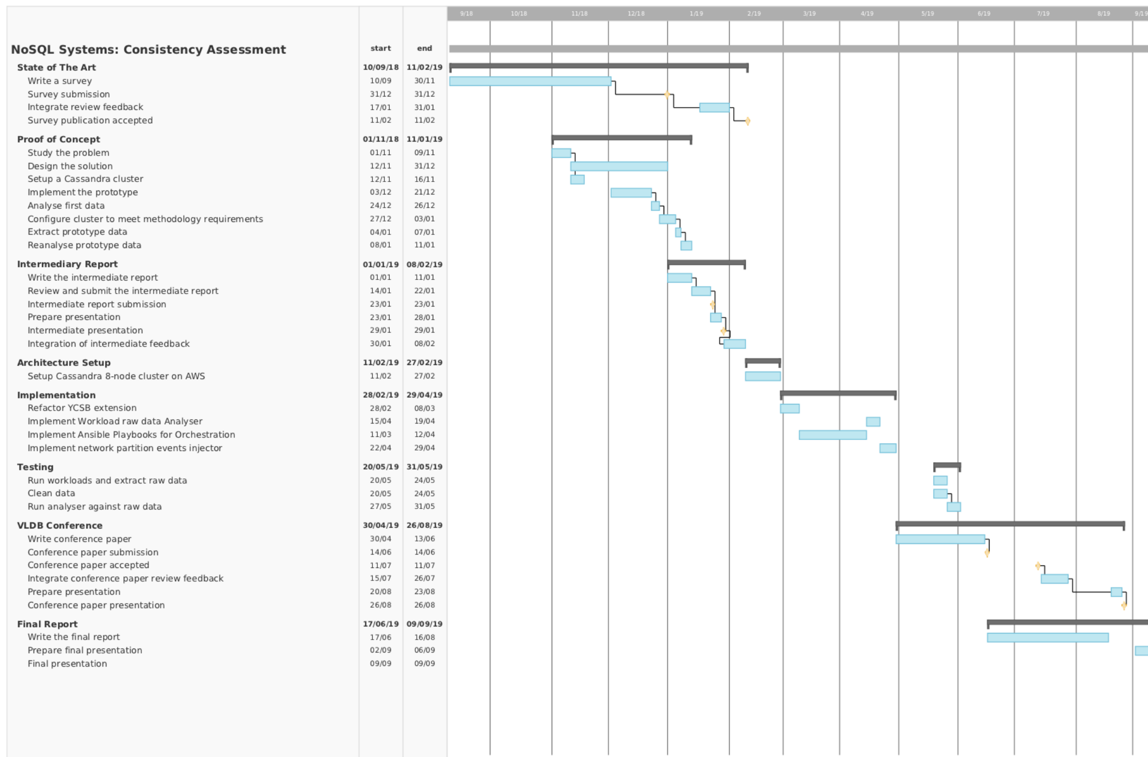


Figure 6.1: Diagram Gantt.

6.1 First Semester

In the first semester, the work focused on writing a survey about Consistency Models and developing a proof of concept of the proposed NoSQL Consistency Benchmark Framework.

6.1.1 Survey

To introduce the research work that supports the current thesis, a survey was written. The focus of this work was to understand how consistency is implemented in five different databases, Redis, Cassandra, MongoDB, Neo4j, and OrientDB while compiling all the related work about consistency benchmarking. Afterward, all the information gathered was compile and discussed into a survey called "Consistency Models of NoSQL Databases". This survey [17] was published in Future Internet Journal and can be found attached in Appendix C.

6.1.2 Proof of Concept

As part of this project, a proof of concept was developed. The architecture setup consists of a four-node Cassandra cluster and a YCSB workload implementation.

6.2 Second Semester

In the second semester, we matured the PoC and implemented the NoSQL Consistency Benchmarking Framework, named CBench-Dynamo, and targeted it to a 8-node Cassandra cluster. Then, we analysed the data generated and conclusions were taken.

6.2.1 Implementation

After understanding that there was an obvious lack of empirical results about relating consistency, availability when subject to network partition events in NoSQL databases, the current project extends the work carried out by Bermbach [13] and matures the work introduced with the Proof of Concept (PoC) implementation developed in the first semester.

6.2.2 Conference Paper

All that we had learned while implementing CBench-Dynamo was compiled into a paper (see Appendix C) that was submitted, accepted and presented in TPCTC 2019, a conference hosted by VLDB 2019 conference in Los Angeles.

Chapter 7

Conclusions and Future Work

This thesis' proposal is one of the first studies that empirically compares the consistency of Redis, Cassandra and MongoDB databases. All of them offer eventual consistency, but each particular implementation has its specificities. Some have the option of offering strong consistency. However, configuring any selected database to favor strong consistency will result in less availability when subject to network partition events, as the CAP theorem preconized.

The following goals of this thesis were achieved:

- Study the state of the art of consistency models;
- Study the state of the art of NoSQL databases;
- Study the state of the art of consistency benchmarking;
- Publish a survey about the investigation carried about the study of the state of the art on consistency models in NoSQL databases;
- Develop a benchmark framework for studying consistency, performance, and availability while subjecting the SUT to network partition events;
- Evaluate how the consistency is affected by the read and write processes of the system, and how the consistency can be improved by tuning system configurations;
- Test this benchmark framework on Cassandra;
- Publish and present a paper at an international conference describing the whole process from designing to testing of the consistency benchmark framework that had been developed.

This thesis started with writing a survey on consistency models. Consistency models can go from Strong to Weak Consistency based on how consistent we want some data to be persisted in a distributed database system. In this survey, it was also addressed a comparison between popular NoSQL databases, Redis, MongoDB, Cassandra, Neo4j, and OrientDB, and how they implement consistency when in distributed configurations. Finally, the same survey stated the problem of a not existing solution on benchmarking consistency in NoSQL databases and that was projected as future work.

The survey enabled us to find a problem, benchmarking consistency on NoSQL databases, and, by referencing the CAP theorem, how consistency correlates with other properties such as availability, network partition tolerance, and performance. Therefore, designing the methodology and by extending the already well-establish performance benchmark framework for NoSQL databases, YCSB, and extending the work carried by Bermbach and Tai [13], this work resulted into CBench-Dynamo, a consistency benchmark for NoSQL databases based on the Dynamo design such as Cassandra.

In weaker consistency models, Cassandra's latency is lower and availability increases, while in stronger consistency models, Cassandra's latency is higher and availability decreases. All this was tested while subjecting the System Under Test to network partition events.

As more NoSQL databases are supported for this framework and implement our methodology, the framework will become a powerful tool on comparing the trade-offs between consistency, latency, availability and network partition tolerance. Hence, providing to database vendors and buyers an easy method to assert which database will better suit their requirements.

References

[1] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," *Proceedings - 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011*, pp. 363–366, 2011.

[2] D. Bermbach, L. Zhao, and S. Sakr, "Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services," in *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking*, vol. 8391. Springer-Verlag, 2013, pp. 32–47.

[3] M. F. Sakr, V. Kolar, and M. Hammoud, "Consistency and Replication – Part II Lecture 11," *Distributed Systems CS 15-440*, 2011.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *Proceedings of the Symposium on Operating Systems Principles*, pp. 205–220, 2007.

[5] K. Bhamra, "A Comparative Analysis of MongoDB and Cassandra," Ph.D. dissertation, Master Thesis. The University of Bergen, 2017. [Online]. Available: http://bora.uib.no/bitstream/handle/1956/17228/kb-thesis.pdf?sequence=1

[6] Ansible, "Ansible Facts Example - Ansible Official Website." [Online]. Available: https://docs.ansible.com/ansible/latest/user{_}guide/playbooks{_}variables.html

[7] E. Marcus, "The myth of the nines," *Blueprints for High Availability*, 2003. [Online]. Available: https://searchstorage.techtarget.com/tip/The-myth-of-the-nines

[8] E. Brewer, "Towards Robust Distributed Systems," pp. 1–12, 2000. [Online]. Available: http://www.eecs.berkeley.edu/{~}brewer/cs262b-2004/PODC-keynote.pdf

[9] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, jun 2002.

[10] D. Pritchett, "BASE: An Acid Alternative," *Queue*, vol. 6, no. 3, pp. 48–55, may 2008.

[11] Yahoo!, "GitHub Repository: YCSB (Yahoo! Cloud Serving Benchmark)." [Online]. Available: https://github.com/brianfrankcooper/YCSB

[12] J. L. Henning, J. A. Arnold, K.-D. Lange, K. Huppler, P. Cao, and J. v. Kistowski, "How to Build a Benchmark," no. September, pp. 333–336, 2015.

[13] D. Bermbach and S. Tai, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pp. 47–56, 2014.

[14] M. Diogo, "GitHub Repository: YCSB-Consistency," 2019.

[15] ——, "GitHub Repository: ansible-dynamo-clusters," 2019. [Online]. Available: https://github.com/migueldiogo/ansible-dynamo-clusters

[16] ——, "GitHub Repository: CBench-Analyser," 2019. [Online]. Available: https://github.com/migueldiogo/cbench-analyser

[17] M. Diogo, B. Cabral, and J. Bernardino, "Consistency Models of NoSQL Databases," *Future Internet*, vol. 11, no. 2, 2019. [Online]. Available: http://www.mdpi.com/1999-5903/11/2/43

[18] P. Viotti and M. Vukolić, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1—-19:34, jun 2016. [Online]. Available: http://doi.acm.org/10.1145/2926965

[19] W. Vogels, "Eventually Consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, jan 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[20] "Read-your-writes (RYW), aka immediate, consistency." [Online]. Available: http://www.dbms2.com/2010/05/01/ryw-read-your-writes-consistency/

[21] J. Likness, "Getting Behind the 9-Ball: Azure Cosmos DB Consistency Levels." [Online]. Available: https://blog.jeremylikness.com/cloud-nosql-azure-cosmosdb-consistency-levels-cfe8348686e6

[22] A. S. Tanenbaum and M. V. M. van Steen, *Distributed Systems: Principles and Paradigms.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[23] Jepsen, "Monotonic Reads." [Online]. Available: https://jepsen.io/consistency/models/monotonic-reads

[24] ——, "Monotonic Writes." [Online]. Available: https://jepsen.io/consistency/models/monotonic-writes

[25] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically Bounded Staleness for Practical Partial Quorums," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 776–787, apr 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212359

[26] ——, "Quantifying Eventual Consistency with PBS," *Commun. ACM*, vol. 57, no. 8, pp. 93–102, aug 2014. [Online]. Available: http://doi.acm.org/10.1145/2632792

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[28] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, jun 2011, pp. 1–5.

[29] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 9:1———9:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038925

[30] H. Wang, J. Li, H. Zhang, and Y. Zhou, "Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra," pp. 71–82, 2014.

[31] Laksham Avinash and Prashant Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, pp. 1–6, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1773922

[32] Datastax, "Data replication." [Online]. Available: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html

[33] ——, "Configuring data consistency." [Online]. Available: https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml{_}config{_}consistency{_}c.html

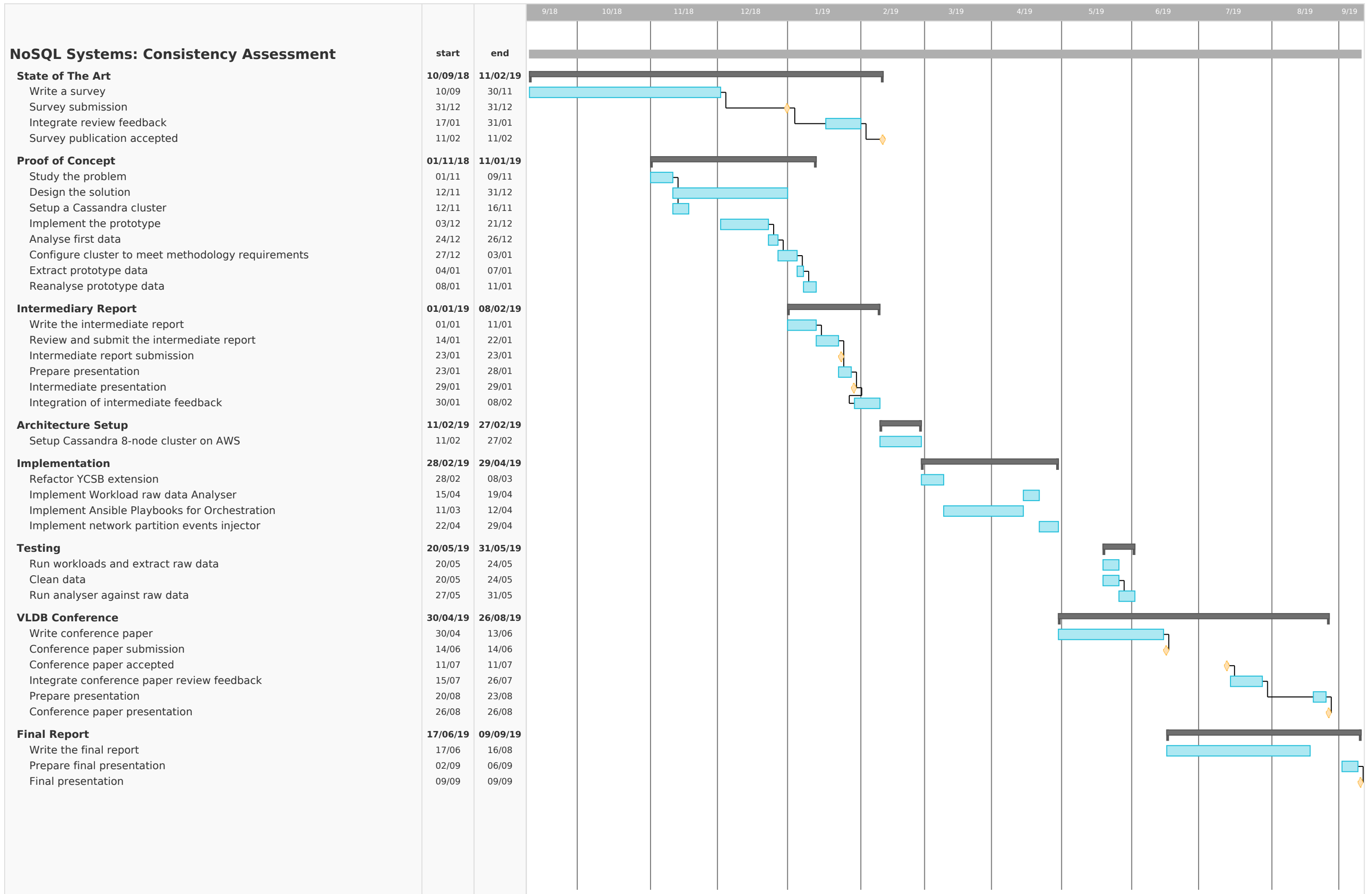
Appendices

This page is intentionally left blank.

Appendix A

Thesis Gantt Diagram.

teamgantt
Created with Free Edition



This page is intentionally left blank.

Appendix B

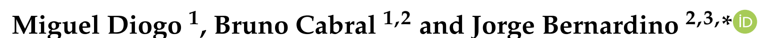
Survey published on Future Internet.

future internet

MDPI

Review

Consistency Models of NoSQL Databases



Miguel Diogo [1], Bruno Cabral [1,2] and Jorge Bernardino [2,3,*]

1. Department of Informatics Engineering, University of Coimbra, Coimbra 3030-290, Portugal; mdiogo@student.dei.uc.pt (M.D.); bcabral@dei.uc.pt (B.C.)
2. Centre of Informatics and Systems of University of Coimbra (CISUC), Coimbra 3030-290, Portugal
3. ISEC—Coimbra Institute of Engineering, Polytechnic of Coimbra, Coimbra 3030-199, Portugal

* Correspondence: jorge@isec.pt

Received: 30 December 2018; Accepted: 11 February 2019; Published: 14 February 2019

Abstract: Internet has become so widespread that most popular websites are accessed by hundreds of millions of people on a daily basis. Monolithic architectures, which were frequently used in the past, were mostly composed of traditional relational database management systems, but quickly have become incapable of sustaining high data traffic very common these days. Meanwhile, NoSQL databases have emerged to provide some missing properties in relational databases like the schema-less design, horizontal scaling, and eventual consistency. This paper analyzes and compares the consistency model implementation on five popular NoSQL databases: Redis, Cassandra, MongoDB, Neo4j, and OrientDB. All of which offer at least eventual consistency, and some have the option of supporting strong consistency. However, imposing strong consistency will result in less availability when subject to network partition events.

Keywords: consistency models; NoSQL databases; redis; cassandra; MongoDB; Neo4j; OrientDB

1. Introduction

Consistency can be simply defined by how the copies from the same data may vary within the same replicated database system [1]. When the readings on a given data object are inconsistent with the last update on this data object, this is a consistency anomaly [2].

For many years, system architects would not compromise when it came to storing data and retrieving it. The ACID (Atomicity, Consistency, Isolation, and Durability) properties were the blueprints for every database management system. Therefore, strong consistency was not a choice. It was a requirement for all systems.

The Internet has grown to a point where billions of people have access to it, not only from a desktop but also from smartphones, smartwatches, and even other servers and services. Nowadays systems need to scale. The "traditional" monolithic database architecture, based on a powerful server, does not guarantees the high availability and network partition required by today's web-scale systems, as demonstrated by the CAP (Consistency, Availability, and Network Partition Tolerance) theorem [3]. To achieve such requirements, systems cannot impose strong consistency.

Traditional relational database architectures usually have a single database instance responding to a few hundred clients. Relational databases implement the strongest consistency model, where each transaction must be immediately committed, and all clients will operate over valid data states. Reads from the same object will present the same value to all simultaneous client requests. Although strong consistency is the ideal requirement for a database, it deeply compromises horizontal-scalability. Horizontal scalability is a more affordable approach when compared to vertical scalability, for enabling higher throughput and the distribution/replication of data across distinct database nodes. On the other hand, vertical scalability relies on a single powerful database server to store data and answer all

requests. Although horizontal scaling may seem preferable, CAP theorem shows that when network partitions occur, one has to opt between availability and consistency [4].

To help solve this problem, NoSQL database systems have emerged. These systems have been created with a standard requirement in mind, scalability.

Some NoSQL databases designers have chosen higher Availability over a more relaxed consistency strategy, an approach known as BASE (Basically Available, Soft-state and Eventually consistent).

The most common NoSQL database systems can be organized into four categories, document databases, column databases, key-value stores, and graph databases. There are also hybrid categories that mix multiple data models known has multi-model databases.

In this work, our goal is to study how consistency is implemented over different non-cloud NoSQL databases. The designers of these database systems have devised different strategies to handle consistency, thus assuming variable tradeoffs between consistency and other quality attributes, such as availability, latency, and network partitioning tolerance.

In this work, we compare the consistency models provided by five of the most popular non-cloud NoSQL database systems [5]. One self-imposed constraint was to select at least one database of each sub-category: Key-value database (Redis); column database (Cassandra); document database (MongoDB), graph database (Neo4j), and multi-model database (OrientDB).

To the best of our knowledge, this is the first study that compares the different consistency solutions provided by the selected NoSQL databases. All of them offer eventual consistency, but each particular implementation has its specificities. Some have the option to offer strong consistency.

The rest of this paper is structured as follows. Section 2 presents the related work. Section 3 presents and discusses consistency models. Section 4 describes the main characteristics of each NoSQL database. In Section 5, we describe and compare the consistency implementations of the five NoSQL databases. Finally, Section 6 presents our conclusions and points out future work.

2. Related Work

Consistency models are analyzed in various works using different assumptions. Bhamra in Reference [6] presents a comparison between the specifications of Cassandra and MongoDB. The author focuses only on a theoretical comparison based on the databases specifications. The objective of this work is to help the reader choosing which database is more suitable for a particular problem. Bhamra starts by making a comparison between Cassandra and MongoDB specifications. Followed by a comparison of the consistency models and, finally, addresses security features, client languages, platform availability, documentation and support, and ease of use. The author compiles the whole comparison into a single table at the end of the article and concludes that MongoDB offers a more versatile approach, querying, and ease of use than Cassandra. The author makes a valuable theoretical analysis. However, it is not presented an experimental evaluation comparing Cassandra and MongoDB databases.

In Reference [7], Han et al. briefly present some NoSQL databases based on the CAP theorem. The authors review and analyze NoSQL databases, such as Redis, Cassandra, and MongoDB and compile the major advantages and disadvantages of these databases. This article concludes that further research is needed to clarify what are the exact limitations of using NoSQL in cloud computing.

Shapiro et al. [2] describe in their work each consistency model. However, the authors do not compare the consistency models against each other by stating that fully implementing each model has not yet been attained because of lack of available frameworks. Shapiro raises three questions from an application point of view. First the robustness of a system versus a specific consistency model. Second, the relation of a model versus a consistency control protocol. The third, and final issue, is to compare consistency models in practice and analyze their advantages and disadvantages. Based on this work, the first two questions are problematic because of the challenge of synthesizing concurrency control from the application specifications.

Pankowski proposes the Lorq algorithm in Reference [8] to balance QoS (Quality of Service) and QoD (Quality of Data). QoS refers to high availability, fault tolerance, and scalability properties. QoD refers to strong consistency. Lorq algorithm is a consensus quorum-based solution for NoSQL data replication. Although this study did not conduct experimental work, the authors state that the Lorq algorithm presents some advantages, for example tools oriented to asynchronous and parallel programming.

Islam and Vrbsky in Reference [9] present two techniques for maintaining consistency and propose a tree-based consistency (TBC) approach. They analyze the advantages and disadvantages of each technique. In the classic approach, in a write request the client needs the acknowledge of every node. While on reading, the system only needs to hit one node. In the quorum approach, in a write request and a read request the system needs to hit only a given number of nodes (quorum) to return a response to the client. In the TBC approach, the system is organized as a tree, where the controller is on the root. The tree defines a path that is used by the replica nodes to propagate the update requests to the replicas (leaves). The authors concluded that the classic approach performs better when write requests represent a low volume; the quorum technique is better to write requests in subsequent read or when write operations are high; the tree-based technique performs better in most cases than the previous two approaches regardless of the request load. Although TBC is an interesting approach, TBC misses abort, commit and rollbacks protocols as the authors have proposed for future work.

Cooper et al. propose in Reference [10], the YCSB (Yahoo! Cloud Serving Benchmark) a benchmarking tool for cloud serving systems. This benchmark fulfills the need for performance comparisons between NoSQL databases and their tradeoffs, such as read performance versus write performance, latency versus durability, and synchronous versus asynchronous replication. The benchmark tiers proposed in this paper include the Tier 1 – Performance and the Tier 2 – Scaling. However, YCSB benchmark lacks tiers, such as Availability, Replication, and Consistency. Although the first two tiers are proposed for future work.

Tudorica and Bucur present a critical comparison between NoSQL systems using multiple criteria [11]. The authors start to introduce multiple taxonomies to classify many NoSQL databases groups, even though there is not an official classification system on this type of databases. They define the following criteria to be used on the theoretical comparison: Persistence, replication, high availability, transactions, rack-locality awareness, implementation, influencers/sponsors, and license type. Tudorica and Bucur concentrate this theoretical comparison into one single table. Afterward, the authors make an empirical performance comparison, between Cassandra, HBase, Sherpa, and MySQL, using YCSB [10]. This article lacks other empirical metrics besides performance, such as consistency.

Wang et al., in Reference [12], present a benchmarking effort on the replication and consistency strategies used in two databases: HBase and Cassandra. Wang et al. motivation are to evaluate tradeoffs, such as latency, consistency, and data replication. The authors conclude that in the latency of read/write operations is hardly improved by adding more replicas to the database. Higher levels of consistency dramatically increase write latency and are not suitable for reading the latest version of data and heavy writes in Cassandra. This paper lacks a more in-depth comparison of how consistency is affected in different configurations. Instead, this work is more focused on studying how consistency levels influence other properties in HBase and Cassandra databases.

Our study is different from all these works by purposing a comparative theoretical analysis of the five of the most popular NoSQL databases in the industry, Redis, MongoDB, Cassandra, Neo4j, and OrientDB, and evaluate how they implement consistency.

3. Consistency Models

In the past, almost all architectures used in databases systems were strong consistent. In these cases, most architectures would have a single database instance only responding to a few hundred clients. Nowadays, many systems are accessed by hundreds of thousands of clients, so there was

a mandatory requirement to system's architectures that scale. However, considering the CAP theorem, high-availability and consistency do conflict on distributed systems when subject to a network partition event. The majority of the projects that have been experiencing such high-traffic have chosen to adopt high-availability over a strong consistent architecture by relaxing the consistency level.

There are two perspectives on consistency, the *data-centric consistency* and the *client-centric consistency*, as illustrated in Figure 1. Data-centric consistency is the consistency analyzed from the replicas' point of view. Client-centric consistency is the consistency analyzed from the clients' point of view [13].

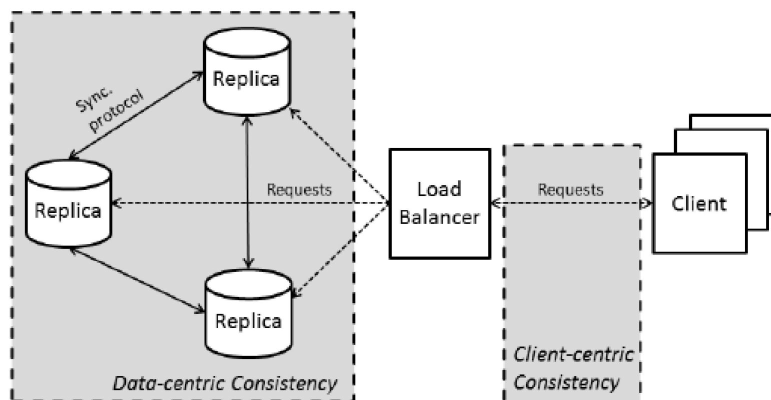


Figure 1. Data-centric and Client-centric consistencies [13].

For both perspectives, there are two dimensions, staleness and ordering. Staleness measures how far from the latest version the return data is. Ordering describes what operations order has been taken in the replica in a data-centric point of view, and, in a client-centric perspective, what order is shown to clients [13]. Figure 2 extends this taxonomy and illustrates how consistency models can be classified by client-centric and data-centric perspectives, and by staleness and ordering dimensions. Under the data-centric perspective we can find two dimensions: *Models for Specifying Consistency* that describe the consistency models that allow measuring and specifying the consistency levels that are tolerable to the application (e.g., Continuous Consistency Model); *Models of Consistent Ordering of Operations* that describe the consistency models that specify what ordering of operations are ensured at the replicas (e.g., Sequential Consistency and Causal Consistency). On the client-centric perspective are also defined two dimensions, *Eventual Consistency* and *Client Consistency Guarantees*. The Eventual Consistency dimension states that all replicas will gradually become consistent if no update operation occurs (e.g., Eventual Consistency Model). The client Consistency Guarantees defines that each client process must ensure some level of consistency while accessing the data value on different replicas (e.g., Monotonic Writes Model, Monotonic Reads Model, Read your Writes Model, and Write Follow Reads).

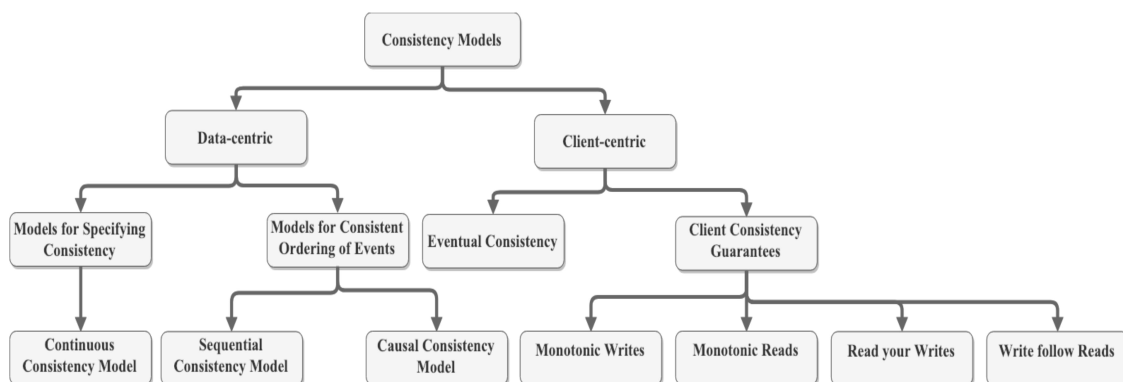


Figure 2. Consistency Models based on Reference [14].

In the next sections, we will review the main consistency models implemented in storage systems: Strong consistency, weak consistency, eventual consistency, causal consistency, read-your-writes consistency, session consistency, monotonic reads consistency, and monotonic writes consistency.

3.1. Strong Consistency

Strong Consistency or Linearization is the strongest consistency model. Each operation must appear committed immediately, and all clients will operate over the same data state. A read operation in an object must wait until the write commits before being able to read the new version. There is also a single global order of events accepted by all storage systems' instances [15].

Strong Consistency leads to a high consistency system, but it compromises scaling by decreasing availability and network partition tolerance.

3.2. Weak Consistency

As the name implies, this model weakens the consistency. It states that a read operation does not guarantee the return of the latest value written. It also does not guarantee a specific order of events [15].

The time period between the write operation and the moment that every read operation returns the updated value is called *the inconsistency window* [16]. This model leads to a highly scalable system because there is no need to involve more than one replica or node into a client request.

3.3. Eventual Consistency

Eventual Consistency strengths the Weak Consistency model. Replicas tend to converge to the same data state. While this convergence process runs, it is possible for read operations to retrieve an older version instead of the latest one. The *inconsistency window* will depend on communication delays between replicas and its sources, the load on the system and the number of replicas involved [16].

This model is half-way a strong consistency model and a weak consistency model. Eventual Consistency is a popular feature offered by many NoSQL databases. Cassandra is one of them, and it can offer availability and network partition on such a level that it does not compromise the usability of the most accessed websites in the world that uses Cassandra. One of them is Facebook, the company that initially developed Cassandra.

3.4. Causal Consistency

If some process updates a given object, all the processes that acknowledge the update on this object will consider the updated value. However, if some other process does not acknowledge the write operation, they will follow the eventual consistency model [16]. Causal consistency is weaker than sequential consistency but stronger than eventual consistency.

Strengthening the Eventual Consistency model to be Causal Consistency decreases availability and network partitioning properties of the system.

3.5. Read-your-writes Consistency

Read-your-writes consistency allows ensuring that a replica is at least current enough to have the changes made by a specific transaction. Because transactions are applied serially, by ensuring a replica has a specific commit applied to it, we know that all transaction commits occurring prior to the specified transaction have also been applied to the replica. If some process updates a given object, this same process will always consider the updated value. Other processes will eventually read the updated value. Therefore, read-your-writes consistency is achieved when the system guarantees that, once a record has been updated, any attempt to read the record will return the updated value [17].

3.6. Session Consistency

If some process makes a request to the storage system in the context of a session, it will follow a read-your-writes consistency model as long as this session exists. Using session consistency, all reads are current with writes from that session, but writes from other sessions may lag. Data from other sessions come in the correct order, just isn't guaranteed to be current. This provides good performance and good availability at half the cost of strong consistency [18].

3.7. Monotonic Reads Consistency

After a process reads some value, all the successive reads will return that same value or a more recent one [19]. In other words, all the reads on the same object by the same process follow a monotonic order. However, this does not guarantee monotonic ordering on the read operations between different processes on the same object. Therefore, monotonic reads ensure that if a process performs read r1, then r2, then r2 cannot observe a state prior to the writes which were reflected in r1; intuitively, reads cannot go backward. Monotonic reads do not apply to operations performed by different processes, only reads by the same process. Monotonic reads can be totally available: Even during a network partition, all nodes can make progress [20].

3.8. Monotonic Writes Consistency

A write operation invoked by a process on a given object needs to be completed before any subsequent write operation on the same object by the same process [19]. In other words, all the writes on the same object by the same process follow a monotonic order. However, this does not guarantee monotonic ordering on the write operations between different processes on the same object. Therefore, monotonic writes ensure that if a process performs write w1, then w2, then all processes observe w1 before w2. Monotonic writes do not apply to operations performed by different processes, only writes by the same process. Monotonic writes can be totally available: Even during a network partition, all nodes can make progress [21].

4. NoSQL Databases Background

In the next sections, we describe succinctly the main characteristics of each one of the five NoSQL databases: Redis, Cassandra, MongoDB, Neo4j, and OrientDB.

4.1. Redis

From the official website, "Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries" [22].

Redis optimizes data in memory by prioritizing high performance, low computation complexity, high memory space efficiency and low application network traffic [23]. Redis guarantees high availability by extending its architecture and introducing the Redis Cluster. Redis on a single instance configuration is strong consistent. In a cluster configuration, Redis is Eventual Consistent when the client reads from the replica nodes.

Redis Cluster requirements are the following [24,25]:

- High performance and linear scalability up to 1000 nodes.
- Relaxed write guarantees. Redis Cluster tries its best to retain all write operations issued by the application, but some of these operations can be lost.
- Availability. Redis Cluster survives network partitions as long as the majority of the master nodes are reachable and there is at least one reachable slave for every master node that is no longer reachable.

4.2. Cassandra

Cassandra is a column NoSQL database [26]. It was initially developed by Facebook to fulfill the needs of the company's Inbox Search services. In 2009, it became an Apache Project.

Cassandra is a database system built with distributed systems in mind, like almost every NoSQL systems out there. Following the CAP theorem, Cassandra will be on the AP (Availability and Network Partition Tolerance) side, hence prioritizing high-availability when subject to network partitioning. As we will further see, Cassandra's consistency can be tuned to be a CP (Consistency and Network Partition Tolerance) database system, so it becomes a strong consistency database when subject to network partitioning.

Cassandra system is a column based NoSQL database [6]. In other words, Cassandra describes data by using columns. A *keyspace* is the outermost container for the entire dataset, corresponding to the entire database, and it is composed of many *column-families*. A column-family represents the same class of objects, like a Car or a Person, and each column-family has different entries of objects called *rows*. Each row is uniquely identified by a *row key* or *partition key* and can hold an arbitrarily large number of columns. A column contains a name-value pair and a timestamp. This timestamp is necessary when solving consistency conflicts.

4.3. MongoDB

MongoDB is a document-based NoSQL database. Its architecture was inspired by the limitations on relational databases like MySQL and Oracle. MongoDB tries to join the best of the RDBMS and NoSQL worlds. From RDBMS, MongoDB took the expressive query language, secondary indexes, strong consistency, and enterprise management while adding NoSQL concepts like dynamic schemas and easier horizontal scalability [27].

MongoDB data model is based on documents. These documents are represented in BSON (Binary JSON). This format extends the well-known JSON (JavaScript Object Notation) to include additional types like int, long, date, and floating point [27].

Documents that represent a similar class of objects are organized as collections. For example, a collection could be the Car collection while a document could be the data item of a single car. Making the analogy with RDBMS, collections are similar to tables. Documents are similar to rows. Fields are similar to columns. Although left-outer JOIN is a valid operation, MongoDB tends to avoid joins by nesting relationships into a single document, like including manufacturer information into a car document [27].

4.4. Neo4j

Neo4j is a graph NoSQL database system. Its data model prioritizes relationships between entities in the form of graphs [28,29].

In the RDBMS world, despite the normalization forms, first introduced in 1970 by Edgar Codd [30], database architects tend to put some extra information into some tables to prevent joins, ending up with several replications of the same data and many consistency problems by having multiple versions of this data. MongoDB also tries to avoid joins by nesting objects which cause the same duplication problem as RDBMS [28].

In Neo4j, a graph is defined by a node and a relationship. As shown in Figure 3, a node represents an entity (i.e., the entity Person). It can have several node attributes. (i.e., the Person with the name "Alice"). Two entities can be linked by a relationship (i.e., the Person with name "Alice" likes the Person with name "Bob"). Relationships can also have properties [31].

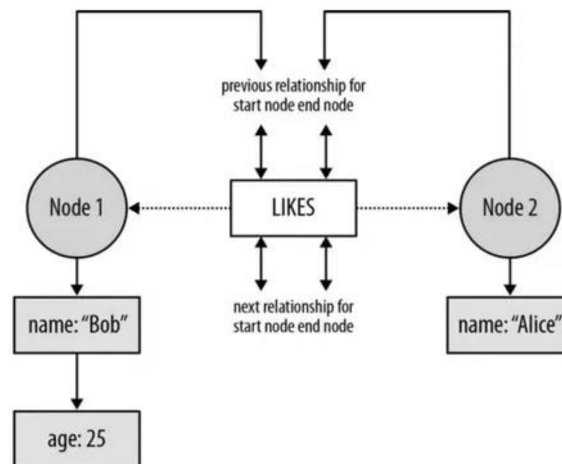


Figure 3. Neo4j Disk Data Structure. Source: Reference [32].

Internally, Neo4j uses linked lists of fixed size record on disk [32]. Properties are stored as a linked list of property records. Each property record holds a key/value. Each node or relationship references its first property record. Relationships are stored in a doubly linked list. A node references its first relationship.

Neo4j is schema optional. It is not necessary to create indexes and constraints. Nodes, relationships, and properties can be created without defining a schema.

Labels define domains by grouping nodes into sets. Nodes that have the same label belongs to the same set. For example, all nodes representing cars could be labeled with the same label: Car. This allows Neo4j to perform operations only within a specific label, such as finding all cars with a given brand.

4.5. OrientDB

OrientDB is a multi-model NoSQL database by mixing more than one model. OrientDB main data models are documents and graphs, but it also implements a key-value engine [29]. This NoSQL database uses the free adjacency list to enable native query processing and it uses document database and object-orientation capabilities to store physical vertices. OrientDB supports schema less, full and mixed modes. Replication and sharding are also supported.

In its Community free edition (Apache 2 License), it does not support features, such as fault tolerance, horizontal scalability, clustering, sharding and replication. However, in its Enterprise paid edition, it supports all the features previously mentioned [29].

A record is the smallest piece of data that can be stored in the database. A record can be a Document, a RecordBytes record (BLOB) a Vertex or even an Edge [33].

Similar to MongoDB's data model, a document is schema-less or schema classes with defined constraints. Documents can easily import or export JSON format [33].

5. NoSQL Consistency Implementations

In this section, we will analyze each consistency implementation in Redis, Cassandra, MongoDB, Neo4j, and OrientDB, NoSQL databases. This review is based on the specifications and focuses on consistency properties. The goal is to understand how each database system scales and how this affects consistency.

5.1. Redis

Redis Cluster distributes keys into 16384 hash slots. Each master stores a subset of the 16384 slots. To determine in which slot a key is stored, the key is hashed using the CRC16 algorithm following by the modulo of 16384:

$$\text{HASH_SLOT} = \text{CRC16(key) mod } 16384$$

However, when we want two keys in the same slot so that we can implement multi-key operations on them, the Redis Cluster implements hash tags. Hash tags ensure that two keys are allocated in the same slot. To achieve this, part of the key has to be a common substring between the two keys and inside brackets. These two keys end up in the same slot because only the substring inside the brackets will be hashed.

For example:

{user:1000}following

{user:1000}followers

Redis Cluster is formed by N nodes connected by TCP connections. Each node has N-1 outgoing connections and N-1 incoming connections. A connection is kept alive as long as the two connected nodes live.

This architecture implements a master-slave model without proxies which means that the application is redirected to the node that has the requested data. Redis nodes do not intermediate responses.

Each master node holds a hash slot. This slot has 1 to N replicas (including the master and its replica nodes). When a master node receives an application issued request, it handles it and asynchronously propagates any changes to its replicas. Then, the master node by default acknowledges the application without an assured replication. This behavior can be overwritten by explicitly making a request using the WAIT command, but this profoundly compromises performance and scalability—the two main strong points of using Redis Cluster.

On the asynchronous replication configuration (default), if the master node dies before replicating and after acknowledging the client, the data is permanently lost. Therefore, the Redis Cluster is not able to guarantee write persistence at all times.

Supposing we have a master node A and a single replica of it representing by A1. If A fails, A1 will be promoted to master, and the cluster will continue to operate. However, if A has no replicas or A and A1 fail at the same time, the Redis Cluster will not be able to continue operating.

In the case of a network partition event, if the client is on the minority side with master A, while on the majority side resides its replicas A1 and A2, if the partition holds for too long (NODE_TIMEOUT) the majority side starts an election process to elect a new master among them, either A1 or A2. Node A is also aware of the timeout and its role change from master to slave. Consequently, it will refuse any further write operations from the client. In this case, Redis Cluster is not the best solution for applications that require high-availability, such as large network partition events.

Supposing that the majority side has N nodes and A and B and its replicas, A1, B1, and B2, respectively, and a network partition event occurs in such way that the replica A1 is separated from the rest. If the partition lasts long enough for assuming A1 as unreachable, Redis Cluster uses a strategy called *replicas migration* to reorganize the cluster and because B has multiple slaves, one of B's replicas will now replicate from A and not from B.

There is also a possibility of reading from replica nodes, instead of from master nodes in order to achieve a more read-scaled system. By using the READONLY command, the client assumes the possibility of reading stale data which is reasonable for situations where having the latest data is not critical. Therefore, leading to an eventual consistency model.

5.2. Cassandra

Cassandra scales up by distributing data across a set of nodes, designated as a cluster. Each node is capable of answering client requests. When a node is working on a client request, it becomes the *coordinator* for that request. It will be responsible for asking to other nodes for the requested data and answering back to the application.

Cassandra partitions data across the cluster by hashing the *row key*. Each node on the ring stores a subset of hashes, in such a way that "the largest hash value wraps around the smallest hash value" [26]. Because of the randomness of the hash functions, data tends to be evenly distributed across the ring.

Replication is the strategy Cassandra uses to achieve a high-available system. Two concepts describe a replication configuration in Cassandra, *replication strategy* and *replication factor* [34]. The *Replication strategy* determines which nodes replicas are placed. The *replication factor* determines how many different nodes have the same data. If the replication factor is R, the node that is responsible for that specific key range copies the data it owns to the next R-1 neighbors, clockwise as in Figure 4.

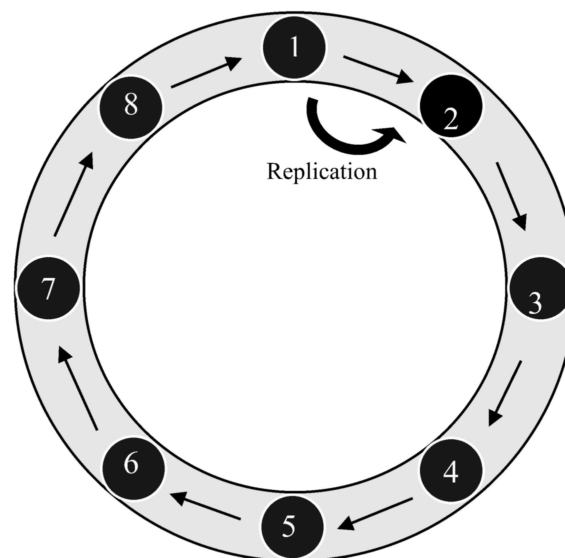


Figure 4. Cassandra Ring.

Cassandra was initially designed to be eventually consistent, high-available and low-latency. However, its consistency can be tuned to match the client's requirements. The following configuration constants describe some of the different write consistency levels [35]:

- *ALL*. Data is written on all replica nodes in the cluster before the coordinator node acknowledges the client. (Strong Consistency, high latency)
- *QUORUM*. Data is written on a given number of replica nodes in the cluster before the coordinator node acknowledges the client. This number is called the quorum. (Eventual Consistency, low latency)
- *LOCAL_QUORUM*. Data is written on a quorum of replica nodes in the same data center as the coordinator node before this last one acknowledges the client. (Eventual Consistency, low latency)
- *ONE*. Data is written in at least one replica node. (Eventual Consistency, low latency)
- *LOCAL_ONE*. Data is written in at least one replica node in the same data center as the coordinator node. (Eventual Consistency, low latency)

Analogous to the write consistency levels, the following configuration constants describe some of the read consistency levels [35]:

- *ALL*. The coordinator node returns the requested data to the client only after all replicas have responded. (Strong consistency, less availability)
- *QUORUM*. The coordinator node returns the requested data to the client only after a quorum of replicas has responded. (Eventual consistency, high-availability)
- *LOCAL QUORUM*. The coordinator node returns the requested data to the client only after a quorum of replicas has responded from the same datacenter as the coordinator. (Eventual consistency, high-availability)
- *ONE*. The coordinator node returns the requested data to the client from the closest replica node. (Eventual consistency, high availability)
- *LOCAL ONE*. The coordinator node returns the requested data to the client from the closest replica node in the local datacenter. (Eventual consistency, high availability)

On the default configuration, Cassandra updates all the replica nodes that have been queried in some reading request to reflect the latest value. This routine is called *Read Repair* and, because a single read triggers it, it puts little stress on the cluster [36]. For reading consistency levels of ONE, the coordinator only asks to one node for the information. Therefore, it cannot *Read Repair* when only one version of the data object is being considered. However, Cassandra has a configuration called *Read Repair Chance*. For instance, given a *Read Repair Change* of 0.1 and a *Replication Factor* of 3, 10% of the reads will trigger a *Read Repair* and hit the three replicas so that the update data is propagated to all three replicas.

Some confusion may raise about the difference between *Replication Factor* and *Write Consistency Level*. The *Replication Factor* does not guarantee that the updated value is fully propagated, only that the data will eventually have a given number of copies in the cluster. The *Write Consistency Level* is responsible for how many copies are made before acknowledging the write operation to the client who had requested it.

To analyze how eventual consistency is affected by the write and read consistency configurations offered by Cassandra, UC Berkeley developed a simulator called Probabilistically Bounded Staleness (PBS) [37]. Figures 5–8, show the resulting curves of our simulation using PBS, given the number of available cluster hosts (N), the read quorum (R) and the write quorum (W). They represent the probability of a client request having the latest version of the data over time (ms) for a given N, W and R combination. All the above configurations assume a *Replication Factor* above 1. If the *Replication Factor* were 1, there would be a single node storing a given data object, therefore the write operation and read operation would only execute in that single node resulting in strong consistency (as seen in Figure 5) for all configurations in Figures 5–8.

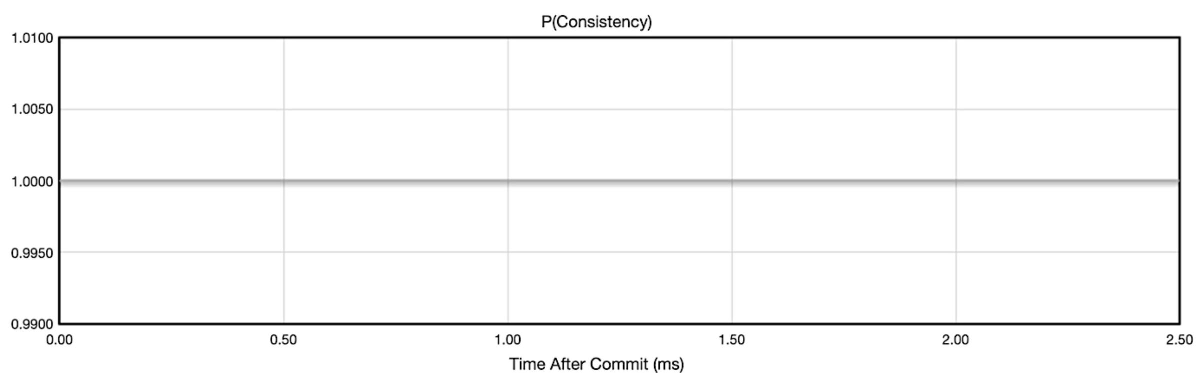


Figure 5. Probabilistically Bounded Staleness (PBS) results for (N = 5, R = 1, W = N = 5) and (N = 5, R = N = 5, W = 1).

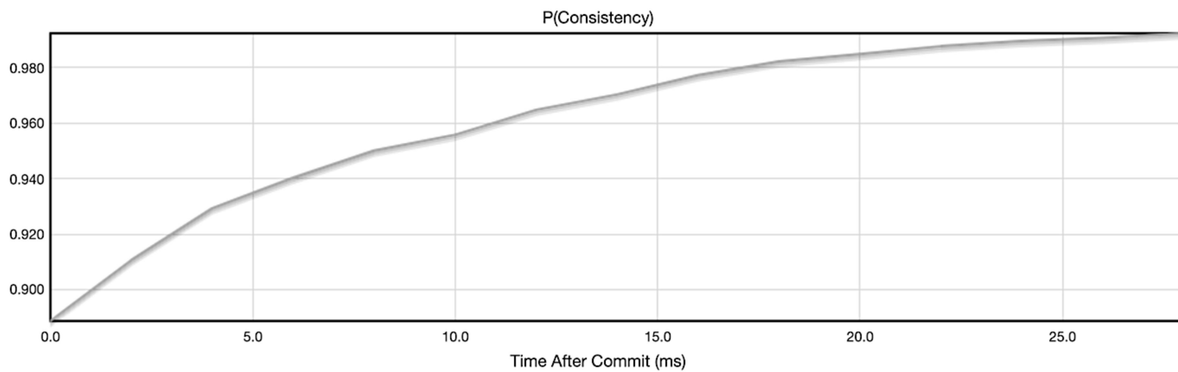


Figure 6. PBS results for (N = 5, R = 1, W = 3).

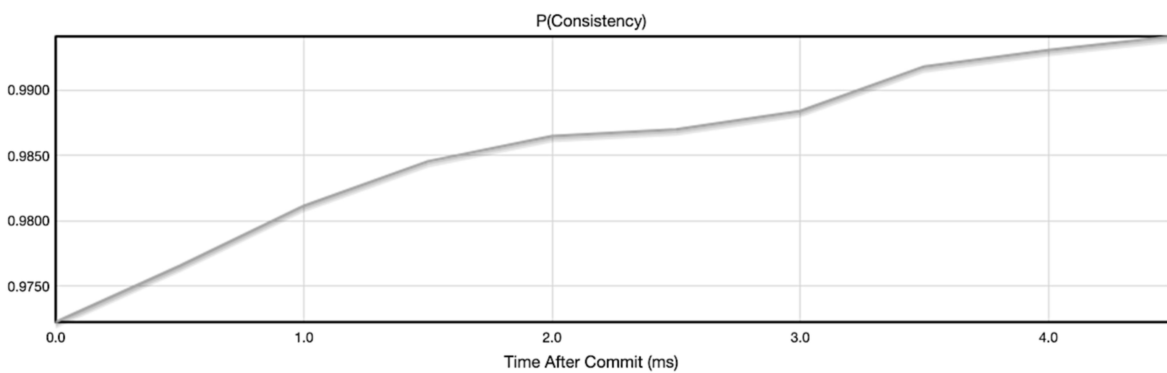


Figure 7. PBS results for (N = 5, R = 3, W = 1).

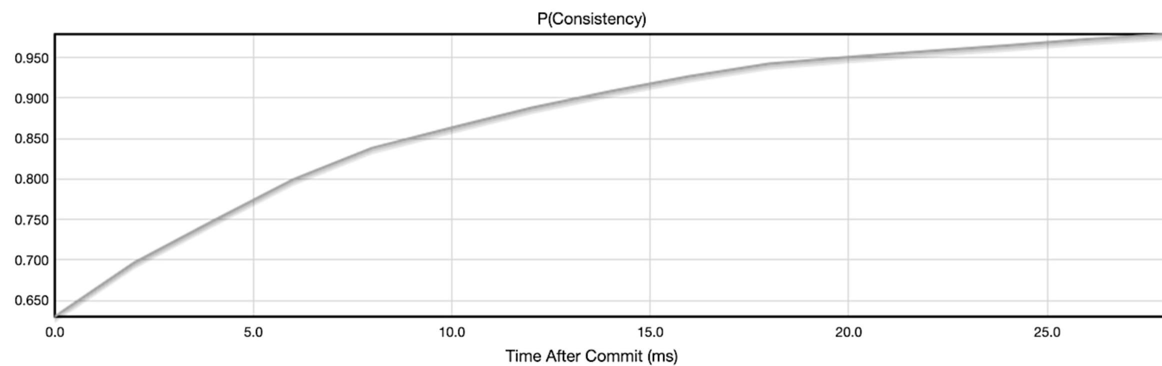


Figure 8. PBS results for (N = 5, R = 1, W = 1).

5.2.1. ALL Write Consistency Level or ALL Read Consistency Level

From Figure 5, we can conclude that the probability of consistency over time is constant, resulting in 100%. That is because each write or read operation is executed on every node available before acknowledging the result to the client. Therefore, this configuration makes the Cassandra cluster strong consistent.

5.2.2. ONE Read Consistency Level and QUORUM Write Consistency Level

In Figure 6, the consistency of a given data object eventually gets to 100%. A write operation needs three updated copies to acknowledge a successful write operation and a read operation returns the first copy the coordinator finds. The time that it is needed to reach 100% consistency is the time that the cluster needs to make all the number of copies previously set on the *Replication Factor*. With Read Consistency Level ONE, Cassandra will depend on the periodically *Read Repair* routines set by the *Read Repair Chance* to update all the copies of the data object and return all the time the same latest version.

5.2.3. QUORUM Read Consistency Level and ONE Write Consistency Level

From Figure 7, we can conclude that the time needed to reach full consistency of a given data object is the shortest of all configurations here (excluding the Figure 5 configuration). Three nodes are approached by the coordinator and the most updated version among them is returned. For each read operation, Cassandra cluster uses its *Read Repair* feature to propagate to all three nodes inside the Quorum (the three nodes), so that they all have the most updated version of the requested data among them. Because *Read Repair* is always triggered by a read, the cluster reaches full consistency faster on the given data object.

5.2.4. ONE Read Consistency Level and ONE Write Consistency Level

In Figure 8, we have the strongest form of eventual consistency configuration in Cassandra. We need just one node with the updated data to acknowledge the write operation. For the reads, the first node the coordinator node chooses will retrieve the requested data. This may or may not be the most updated version of the data object. Eventually, the most updated version will be returned on all requests. The time needed to get to a 100% probability of consistency will depend on the *Read Repair Chance* and the *Replication Factor*. The higher the probability of the *Read Repair Chance*, the shorter the time to get to full consistency. The lower the *Replication Factor*, the shorter the time to get to full consistency. Modifying the *Read Repair Chance* and the *Replication Factor* to reach consistency faster will result in higher latencies because more copies and nodes are involved in the read and write operations for each client request.

5.3. MongoDB

One of the strongest features of MongoDB is horizontal scaling by using a technique called sharding. Sharding allows distributing data across many data nodes. Hence, avoiding the architectures composed of a couple of big and powerful machines. MongoDB balances data across these nodes in an automatic way. There are three types of sharding:

Range-based Sharding: documents are distributed based on their shard key-values. Consequently, two shard key-values close to each other are likely to be on the same shard. Hence, optimizing operations between them.

Hash-based Sharding: the key-values are subject to an MD5 hash. This sharding strategy tends to distribute data across shards uniformly. Although, it performs worse in range-based queries.

Location-aware Sharding: the user can specify a custom configuration to accomplish application requirements. For example, high-demanding data can be stored In-Memory (Enterprise Edition), and less popular data can be stored on the disk.

A dispatcher called Query Router will redirect application issued queries to the correct shard depending on the sharding strategy and shard value.

MongoDB follows the ACID (Atomicity, Consistency, Isolation, and Durability) properties [38] similar to RDBMS implementations:

- **Atomicity**. MongoDB supports single operation inserts and updates;
- **Consistency**. MongoDB can be used on a strong consistency approach;
- **Isolation**. While a document is updated, it is entirely isolated. Any error would result in a rollback operation, and no user will be reading stale data;
- **Durability**. MongoDB implements a feature called write concern. Write concern are user-defined policies that need to be fulfilled in order to commit (i.e., writing at least three replicas before commit).

MongoDB allows configuring a replica set. A replica set has primary replica set members and secondary replica set members. There are two configurations based on the desired consistency level:

- **Strong consistency**. Applications write and read from the primary replica set member. The primary member will write all the operations that made it transact to the new state. These operations are idempotent and constitute the oplog (operations log). After the primary member acknowledges the application of the committed data and operations logging, secondary replica set members can now read from this log and replay all operations so that they can be on the same state of the primary member.
- **Eventual consistency**. Applications can read from secondary replica set members if they do not prioritize reading the latest data.

In the case of a primary member failover, secondary replicas will elect a new primary among them by using the Raft consensus algorithm [39]. Once the primary member is elected, it will be responsible for updating the oplog read by secondary members. In the case of a recovery of the primary member, this will play the role of a secondary member for then on.

Oplog has a configurable back-limit history (default: 5% of the available disk space). If a secondary member fails longer enough to need operations that are no longer available in the oplog, all the databases, collections, and indexes directives are copied from the primary member or another secondary member. This process is called *initial synchronization*. The same one that is used when adding a new member to the replica set

5.4. Neo4j

Neo4J is considered the most popular graph database worldwide, used in areas, such as health, government, automotive production, military area and others. Neo4j favors strong consistency and availability. Neo4j has clustering features in its Enterprise Edition. These features are capable of providing a fault tolerant platform, reading scale up and Causal Consistency model.

A cluster is composed of two types of nodes, core servers and read replicas. Core Servers' primary responsibility is to ensure data durability. Once the majority of the Core Servers set has accepted a given transaction, the client will be acknowledged of the commit. In order to calculate the number of Core Servers required to tolerate F failed servers, Neo4j states that the number of Core Servers needed is 2F + 1. In a real situation where occurs a certain number of failed Core Servers greater than F, the cluster will become read-only to preserve data safety, because the minimum requirements to achieve write consensus has been compromised.

Figure 9 shows Neo4j Cluster Architecture. Read replicas' main responsibility is to ease the load from read requests. They asynchronously reflect the changes consented by the majority of the Core Servers set. As the Read Replicas do not change data states, they can view as disposable servers, which means that their arrival or departure will only decrease or increase query latency respectively, but it will never compromise data availability [40].

Neo4j in a single node architecture is strongly consistent. In Neo4j Enterprise Edition, the cluster ensures causal consistency. As we previously mentioned, causal consistency guarantees that reading data previously written from the same client will be consistent. However, we have eventual consistency when reading data that was changed by other clients, because there is a millisecond time window which the latest data has not been propagated yet [40]. Neo4j is located on the CA quadrant by providing consistency and availability.

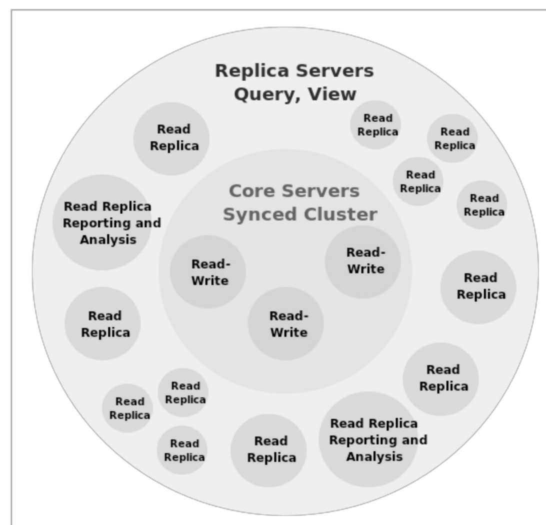


Figure 9. Neo4j Cluster Architecture.

5.5. OrientDB

OrientDB has a master-slave strategy to achieve a more scalable architecture. Every time a client makes a request, it would have to hit a single master. This master would then propagate any state change to its replicas. Finally, it would acknowledge the client. This approach made the database strong consistent, and it was only scaling the reads because there was only a single master node, which represented a severe bottleneck in the system.

Some years ago, OrientDB announced a new paradigm shift, the multi-master architecture, with the premise that all nodes must accept writes. In the default configuration, a client acknowledges only after the majority of the master nodes commit the new date state. Then, asynchronously, the master nodes that have not committed are fixed, and the data propagates to the replicas (read-only). This new approach made the system eventual consistent when reading from an unfixed master or some replica that has not yet received the latest data. However, if the same master is hit over and over again, the client will get strong consistency, while compromising performance. OrientDB handles this with three client load balancing configurations:

- **STICKY**. The default configuration. The client remains connected to the same server until the database closes. (Strong Consistency, high latency)
- **ROUND_ROBIN_CONNECT**. The client connects to a different server at each connection following a round robin schedule [41]. (Session Consistency)
- **ROUND_ROBIN_REQUEST**. The client connects to a different server at each request following a round robin schedule [41]. (Eventual Consistency, low latency)

Clients have the ability to know whether the version of the data retrieved is updated [42]. OrientDB supports Multi-Version Concurrency Control (MVCC) with atomic operations. This avoids the use of locks in the server. Every time a read request is made, if the version of the data is equal to the one that is on the response payload addressed to the client, the operation is successful. Otherwise, OrientDB generates an error that can be handled by the client.

5.6. Summary

In this section, we compared the different consistency implementations of several NoSQL databases. Table 1 summarizes the consistency models supported by the five NoSQL databases: Redis, Cassandra, MongoDB, Neo4j, and OrientDB.

Table 1. Consistency models comparison.

	Strong	Casual	Session	Eventual	Weak
Redis		[2]		✓ [1]	
Cassandra	✓			✓ [1]	
MongoDB	✓ [1]	✓	✓	✓	
Neo4j	✓ [1]	✓		✓	
OrientDB	✓ [1]		✓	✓	

[1] default configuration, [2] across replicas.

Redis is a high-available and very low latency (the best of the group) database, due to the in-memory architecture. It can relax consistency to an eventual consistency level.

Cassandra introduces a storing system composed of many nodes on a ring. Cassandra is the database system that offers the broadest spectrum of eventual consistency levels. Cassandra is cheaper for storing data than Redis and has a robust eventual consistency architecture while maintaining high-availability and low-latency. Cassandra's best uses cases are core data storage from applications that don't always need the latest data but prefer a high available and low latency service instead.

MongoDB is the database system that offers more consistency configurations, strong consistency, causal consistency, session consistency, and eventual consistency. Although MongoDB's default consistency model is strong consistency, it has the ability to perform at higher availability and lower latency when on its eventual consistency configuration. However, MongoDB is a single master architecture. For this reason, it does not scale writes well as it scales stale data reads (eventual consistency). MongoDB's best use cases are, for example, logging and data that don't demand write requests from a large pool of clients.

Neo4j promotes consistency and availability. Neo4j does not support data partitioning. However, it has a variant of the master-slave model, where it is possible to read from replicas that may or may not have the latest data. Therefore, the applications can choose to have eventual consistent readings.

Similar to MongoDB and Neo4j, OrientDB allows the application to choose which consistency level it prefers. OrientDB defaults to strong consistency when its data is read solely from master nodes and eventual consistency when reading from replicas. In their eventual consistency configurations, Neo4j's and OrientDB's scaling limitations are similar to the ones found on MongoDB.

Figure 10 summarizes the lessons learned by depicting where each database positions itself with respect to the three quality attributes addressed by the CAP theorem. Note that this analysis only considers the default configurations of each database engine [43]. The three vertices of the theorem describe each property, **C**onsistency, **A**vailability and Network **P**artition Tolerance. Neo4j, OrientDB, and Relational DBs favor strong consistency and availability. Cassandra favors eventual consistency, resulting in high availability, better tolerance to network partition and low latency. Finally, MongoDB and Redis favor strong consistency and network partition tolerance.

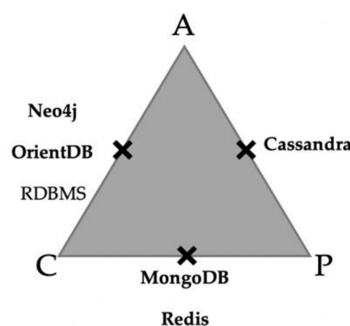


Figure 10. Consistency, Availability, and Network Partition Tolerance (CAP) Theorem and classification of databases based on their default configurations.

6. Conclusions and Future Work

In this work, we studied the consistency models implemented by five popular NoSQL database systems: Redis, Cassandra, MongoDB, Neo4j, and OrientDB. Configuring any selected database to favor strong consistency will result in less availability when subject to network partition events, as the CAP theorem preconized. When considering the default consistency model in each distributed database, it is clear that to be partition tolerant and ensure high consistency, MongoDB is the preferable option. But, if one wants to provide high availability, Cassandra is the better choice. Whenever partition intolerance or non-distributed databases are an option, both Neo4j and OrientDB are able to offer high consistency.

As future work, we propose to do an empirical evaluation to study, compare and better understand the impact of different consistency solutions and configurations on the selected NoSQL databases. Consequently, comparing the consistency models in practice, but for the rest of the NoSQL spectrum, so that we understand their pros and cons. We also intend to evaluate the real impact of the different consistency models over the other quality attributes considered on the CAP theorem.

Author Contributions: Conceptualization, J.B. and B.C; Methodology, B.C. and J.B.; Software, M.D.; Validation and Formal Analysis, M.D.; Investigation, M.D, B.C. and J.B; Resources, M.D.; Data Curation, M.D.; Writing-Original Draft Preparation, M.D.; Writing-Review and Editing, J.B and B.C.; Visualization, M.D.; Supervision, J.B and B.C.; Project Administration, J.B and B.C.; Funding Acquisition, J.B.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Miret, L.P. Consistency Models in Modern Distributed Systems. An Approach to Eventual Consistency. Master's Thesis, Universitat Politècnica de València, València, Spain, 2015.
2. Shapiro, M.; Sutra, P. Database Consistency Models. In *Encyclopedia of Big Data Technologies*; Springer: Cham, Switzerland, 2018; pp. 1–11.
3. Brewer, E. Towards Robust Distributed Systems. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, Portland, OR, USA, 16–19 July 2000; pp. 1–12.
4. Gilbert, S.; Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News* **2002**, *33*, 51–59. [CrossRef]
5. DB-Engines Ranking. Available online: https://db-engines.com/en/ranking (accessed on 30 December 2018).
6. Bhamra, K. A Comparative Analysis of MongoDB and Cassandra. Master's Thesis, The University of Bergen, Bergen, Norway, 2017.
7. Han, J.; Haihong, E.; Le, G.; Du, J. Survey on NoSQL database. In Proceedings of the 2011 6th International Conference on Pervasive Computing and Applications (ICPCA 2011), Port Elizabeth, South Africa, 26–28 October 2011; pp. 363–366.
8. Pankowski, T. Consistency and availability of Data in replicated NoSQL databases. In Proceedings of the 2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Barcelona, Spain, 29–30 April 2015; pp. 102–109.
9. Islam, A.; Vrbsky, S.V. Comparison of consistency approaches for cloud databases. *Int. J. Cloud Comput.* **2013**, *2*, 378–398. [CrossRef]
10. Yahoo! YCSB (Yahoo! Cloud Serving Benchmark). Available online: https://github.com/brianfrankcooper/YCSB (accessed on 30 December 2018).
11. Tudorica, B.G.; Bucur, C. A comparison between several NoSQL databases with comments and notes. In Proceedings of the 2011 RoEduNet International Conference 10th Edition: Networking in Education and Research, Iasi, Romania, 23–25 June 2011; pp. 1–5.
12. Wang, H.; Li, J.; Zhang, H.; Zhou, Y. Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 4th and 5th Workshops*; Springer: Cham, Switzerland, 2014; Volume 8807, pp. 71–82.

13. Bermbach, D.; Zhao, L.; Sakr, S. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking*; Springer: Cham, Switzerland, 2013; Volume 8391, pp. 32–47.
14. Sakr, M.F.; Kolar, V.; Hammoud, M. Consistency and Replication—Part II Lecture 11. Available online: https://web2.qatar.cmu.edu/~{}msakr/15440-f11/lectures/Lecture11_15440_VKO_10Oct_2011.pptx (accessed on 30 December 2018).
15. Viotti, P.; Vukolić, M. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* **2016**, *49*, 19. [CrossRef]
16. Vogels, W. Eventually Consistent. *Commun. ACM* **2009**, *52*, 40–44. [CrossRef]
17. Read-Your-Writes (RYW), Aka Immediate, Consistency. Available online: http://www.dbms2.com/2010/05/01/ryw-read-your-writes-consistency/ (accessed on 30 December 2018).
18. Likness, J. Getting Behind the 9-Ball: Azure Cosmos DB Consistency Levels. Available online: https://blog.jeremylikness.com/cloud-nosql-azure-cosmosdb-consistency-levels-cfe8348686e6 (accessed on 30 December 2018).
19. Tanenbaum, A.S.; van Steen, M.V.M. *Distributed Systems: Principles and Paradigms*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 2006.
20. Jepsen. Monotonic Reads. Available online: https://jepsen.io/consistency/models/monotonic-reads (accessed on 30 December 2018).
21. Jepsen. Monotonic Writes. Available online: https://jepsen.io/consistency/models/monotonic-writes (accessed on 30 December 2018).
22. Redis Labs. Redis Website. Available online: https://redis.io/ (accessed on 30 December 2018).
23. Joshi, L. Do You Really Know Redis? *Redis Labs White Pap.* Available online: https://redislabs.com/docs/really-know-redis (accessed on 11 February 2019).
24. Redis Labs. Redis Cluster Tutorial. 2018. Available online: https://redis.io/topics/cluster-tutorial (accessed on 30 December 2018).
25. Redis Labs. Redis Cluster Specs. 2018. Available online: https://redis.io/topics/cluster-spec (accessed on 30 December 2018).
26. Avinash, L.; Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40.
27. MongoDB Inc. MongoDB Architecture Guide; MongoDB White Pap. Available online: https://www.mongodb.com/collateral/mongodb-architecture-guide (accessed on 30 December 2018).
28. Neo4j. Overcoming SQL Strain and SQL Pain; Neo4j White Pap. Available online: https://neo4j.com/resources-old/overcoming-sql-strain-white-paper (accessed on 30 December 2018).
29. Fernandes, D.; Bernardino, J. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In Proceedings of the 7th International Conference on Data Science, Technology and Applications, {DATA} 2018, Porto, Portugal, 26–28 July 2018; pp. 373–380.
30. Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* **1970**, *13*, 377–387. [CrossRef]
31. Hunger, M.; Boyd, R.; Lyon, W. The Definitive Guide to Graph Databases for the RDBMS Developer; Neo4j White Pap. Available online: https://neo4j.com/whitepapers/rdbms-developers-graph-databases-ebook (accessed on 30 December 2018).
32. Neo4J Internals. Available online: https://www.slideshare.net/thobe/an-overview-of-neo4j-internals (accessed on 30 December 2018).
33. Callidus Software. OrientDB Manual—Version 2.2.x. Available online: https://orientdb.com/docs/2.2.x/ (accessed on 30 December 2018).
34. Datastax. Data Replication. Available online: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html (accessed on 30 December 2018).
35. Datastax. Configuring Data Consistency. Available online: https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html (accessed on 30 December 2018).
36. Datastax. Read Repair: Repair during Read Path. Available online: https://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html (accessed on 30 December 2018).
37. Probabilistically Bounded Staleness (PBS) Simulator. Available online: http://pbs.cs.berkeley.edu/ (accessed on 11 February 2019).

38. Haerder, T.; Reuter, A. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.* **1983**, *15*, 287–317. [CrossRef]
39. Ongaro, D.; Ousterhout, J. In Search of an Understandable Consensus Algorithm. In Proceedings of the: 2014 USENIX Annual Technical Conference (USENIX ATC '14), Philadelphia, PA, USA, 19–20 June 2014; Volume 22, pp. 305–320.
40. Neo4j. Chapter 4. Clustering. Enterprise Edition. Neo4j Online Docs. Available online: https://neo4j.com/docs/operations-manual/current/clustering/ (accessed on 11 February 2019).
41. Arpaci-Dusseau, R.H.; Arpaci-Dusseau, A.C.; Assumptions, W. Operating Systems: Three Easy Pieces. In *Operating Systems: Three Easy Pieces, 0.91*; Arpaci-Dusseau Books; CreateSpace Independent Publishing: Scotts Valley, CA, USA, 2015; pp. 7–9.
42. OrientDB Concurrency Docs. Available online: https://orientdb.com/docs/2.2.x/Concurrency.html (accessed on 11 February 2019).
43. Lourenço, J.; Cabral, B.; Carreiro, P.; Vieira, M.; Bernardino, J. Choosing the right NoSQL database for the job: A quality attribute evaluation. *J. Big Data* **2015**, *2*, 18–44. [CrossRef]

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).

Appendix C

Conference paper presented on TPCTC/VLDB 2019 conference.

CBench-Dynamo: A Consistency Benchmark for NoSQL Database Systems

Miguel Diogo[1[0000-0003-2567-5103]], Bruno Cabral[2[0000-0001-9699-1133]]

and Jorge Bernardino[3,2[0000-0001-9660-2011]]

[1] Universidade de Coimbra – Departamento de Informática Coimbra, Portugal
[2] Centre of Informatics and Engineering of University of Coimbra - CISUC
[3] Polytechnic of Coimbra – ISEC, Coimbra, Portugal
mdiogo@student.dei.uc.pt
bcabral@dei.uc.pt
jorge@isec.pt

Abstract. Nowadays software architects face new challenges because Internet has grown to a point where popular websites are accessed by hundreds of millions of people on a daily basis. One powerful machine is no longer economically viable and resilient in order to handle such outstanding traffic and architectures have since been migrated to horizontal scaling. However, traditional databases, usually associated with a relational design, were not ready for horizontal scaling. Therefore, NoSQL databases have proposed to fill the gap left by their predecessors. This new paradigm is proposed to better serve currently massive scaled-up Internet usage when consistency is no longer a top priority and a high available service is preferable. Cassandra is a NoSQL database based on the Amazon Dynamo design. Dynamo-based databases are designed to run in a cluster while offering high availability and eventual consistency to clients when subject to network partition events. Therefore, our goal is to propose CBench-Dynamo, the first consistency benchmark for NoSQL databases. Our proposed benchmark correlates properties, such as performance, consistency, and availability, in different consistency configurations while subjecting the System Under Test to network partition events.

Keywords: Consistency, Availability, Network Fault Tolerance, NoSQL Databases, Benchmark, Dynamo, Cassandra.

1 Introduction

The Internet has grown to a point where billions of people have access to it, not only from a desktop but also from smartphones, smartwatches, and even other servers and services. Nowadays systems need to scale. The monolithic database architecture, based on a powerful server, does not guarantee the high availability and network partition required by today's web-scale systems, as demonstrated by the CAP (Consistency, Availability, and Network Partition Tolerance) theorem [1]. Strong consistency is a property that has been relaxed to achieve a more scalable database system. Relational databases foundations were designed to support strong consistency. Each transaction

must be immediately committed, and all clients will operate over consistent data states. Reads from the same object will present the same value to all client requests. Although strong consistency is the ideal requirement for a database, it deeply compromises horizontal-scalability. Horizontal scalability is a more affordable approach when compared to vertical scalability. It enables higher throughput and data distribution across multiple database nodes. On the other hand, vertical scalability relies on a single powerful database server to store data and answer all requests. Although horizontal scaling may seem preferable, CAP theorem presents that when network partitions occur, one has to opt between availability and consistency [2]. Horizontal scaling has inspired a new category of databases called NoSQL. These systems have been created with a common requirement in mind, scalability. Several NoSQL designs prioritize high-availability over a more relaxed consistency strategy, an approach known as BASE (Basically Available, Soft-state and Eventually consistent) [3].

Although performance frameworks, such as YCSB [4], have been developed for benchmarking NoSQL databases, they lack a consistency tier to fully compare the tradeoffs annunciated by the CAP theorem.

In this work, we propose CBench-Dynamo, a benchmark for testing consistency and availability on a horizontal-scaled system. We also define how to address the main quality attributes of a benchmark, i.e. Relevance, Reproducibility, Fairness, Verifiability, and Usability [5]. Our goal is to extract different measurements on performance, consistency, and availability with different consistency configurations of the System Under Test (SUT) while subjecting this system to network partition events.

Finally, we will run the proposed benchmark on a Cassandra cluster and discuss the resulting measurements.

2 Related Work

Cooper et al. [6] propose the YCSB (Yahoo! Cloud Serving Benchmark), a benchmarking framework for cloud serving systems. It comes to fill the need for performance comparisons between NoSQL databases and keep tracking of their tradeoffs such as read performance versus write performance, latency versus durability, and synchronous versus asynchronous replication. Although benchmark tiers such as performance and scaling are included in YCSB, it lacks other tiers such as availability and consistency.

Bailis et al. [7] suggest an approach that predicts the expected consistency of an eventually consistent Dynamo data store using models the authors developed called Probabilistically Bounded Staleness (PBS). This approach lacks a benchmark framework.

The work of Bermbach and Tai [8] propose a benchmark methodology on Amazon's cloud database AWS S3. This is the closest work of what we aim to do in this paper. Bermbach and Tai project a long-term monitor system on AWS S3 to evaluate how this service changes its consistency ability over time and the benchmark approach used can be easily extended for other usages and databases as we aim to demonstrate in our paper.

Patil et al. [9] propose a benchmark architecture that evaluates time to consistency. The authors extend the YCSB framework and add support to distributed architectures by using ZooKeeper for coordination. However, since 2011, development of the YCSB++ framework has been discontinued, and Cassandra support is still in progress. Only HBase and Acumulo support are available, but they are outdated as major releases of both databases have been released. YCSB++ also does not fully evaluate consistency trade-offs based on the CAP theorem as YCSB++ does not support network partition events.

Our work differentiates from the rest by proposing the first NoSQL consistency benchmarking framework and testing it on Cassandra. To the best of our knowledge, no such framework compares consistency levels to other quality attributes such as availability and performance while subjecting the target system to network partition events.

3 Dynamo

Dynamo design and implementation were first introduced by Amazon as a highly available key-value storage system [10]. Since then, Amazon has built many cloud services emerged around this design, e.g. Amazon DynamoDB and Amazon S3.

Dynamo prioritizes eventual consistency, targeted to applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes.

Dynamo was designed to scale incrementally, hence it was designed with a partition mechanism in mind. Dynamo's partition mechanism is based on a consistent hashing to distribute the load across multiple data nodes. The output of this hashing function can be illustrated as a ring as seen in

Fig. 1, in which the highest output wraps around the smallest one. Each node is assigned a random value within the range of the hashing function. To know which node will store a given data value, the correspondent key of this value is hashed. Then, we walk the ring clockwise, from the smallest to the largest number, to find the first node with a position larger than the hashing result.

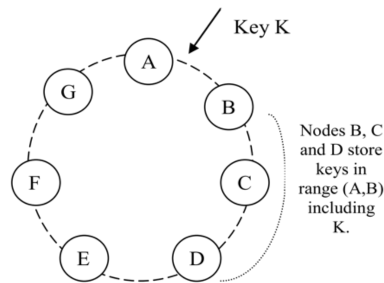


Fig. 1. Partitioning and replication of keys in Dynamo ring [10].

4 Consistency in Dynamo-based Databases

Amazon's Dynamo based databases such as Cassandra, all use the same variant of quorum-style replication [11]. Quorum-style replication is associated with a replication factor *N,* i.e. the number of replicas that some data eventual will be in. Read and write consistency can be configured as follows, *ONE*, *QUORUM*, or *ALL*.

The following configurations describe the differences between the three write consistency levels for Dynamo-based database systems [12]:

- *ALL*: data is written on all replica nodes in the cluster before the coordinator node acknowledges the client. Therefore, this configuration has: *Strong Consistency* and *High latency*.
- *QUORUM*: data is written on a given number of replica nodes in the cluster before the coordinator node acknowledges the client, where this number is called the quorum. This configuration has: *Eventual Consistency* and *Low latency*.
- *ONE*: data is written in at least one replica node. This configuration has: *Eventual Consistency* and *Low latency*.

Analogous to the write consistency levels, the following configuration constants describe the differences between the three read consistency levels for Dynamo-based database systems [12]:

- *ALL*. The coordinator node returns the requested data to the client only after all replicas have responded. This configuration has: *Strong consistency* and *Less availability*.
- *QUORUM*. The coordinator node returns the requested data to the client only after a quorum of replicas has responded. This configuration has: *Eventual consistency* and *High-availability*.
- *ONE*. The coordinator node returns the requested data to the client from the closest replica node. This configuration has: *Eventual consistency* and *High availability*.

Under normal operation, i.e. without network partition events, given the number of replicas required for a read operation as *R,* the number of replicas required for a write operation as *W*, and the replication factor as *N*, Dynamo-based databases guarantee consistency when [11]:

$$R + W > N \tag{1}$$

Given R_{Quorum} and W_{Quorum}, as Read Consistency and Write Consistency set to *QUORUM*, respectively, and the *floor* function that takes as input a real number and round it down to the closest integer. The conversion from the *QUORUM* notation to the *R*, *W* notation is as follows:

$$R_{QUORUM}, W_{QUORUM} = R, W = floor\left(\frac{N}{2} + 1\right) \tag{2}$$

Under abnormal operations where a network partition event had occurred, if consistency is set to strong, i.e. *ALL*, availability is compromised as the Fig. 2 illustrates.

The scenario illustrated by Fig. 2 describes a situation where a read operation needs to involve the total number of replicas *N* in order to retrieve the data to the client. In case of a network partition, e.g. *node C* crashes, the coordinator of the request, *node A*, was not able to serve data, hence the total number of replicas had been involved in the operation, and the coordinator had no other option than announcing a lack of service availability to the client, resulting in a *TIMEOUT* response.

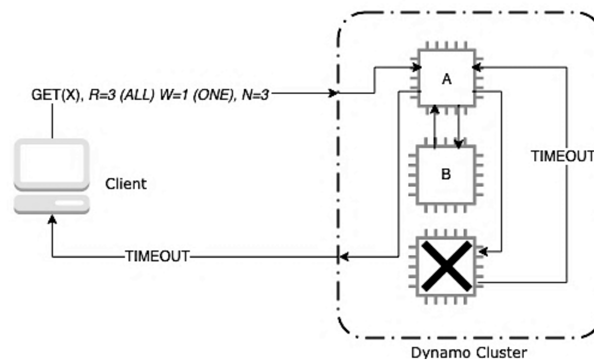


Fig. 2. Data request on abnormal operation where a node fails, and strong read consistency is set (i.e. *ALL*).

Under abnormal operations where a network partition event had occurred, if consistency is set to eventual configuration *(ONE)*, we achieve service availability even in the presence of a node crash as the Fig. 3 illustrates. The scenario illustrated by Fig. 3 describes a situation where a read operation only needs to involve one replica in order to retrieve the requested data to the client. Because only one replica had been involved, the response may not contain the latest data as this example suggests. Although consistent responses are not ensured, this configuration results in a high-available service.

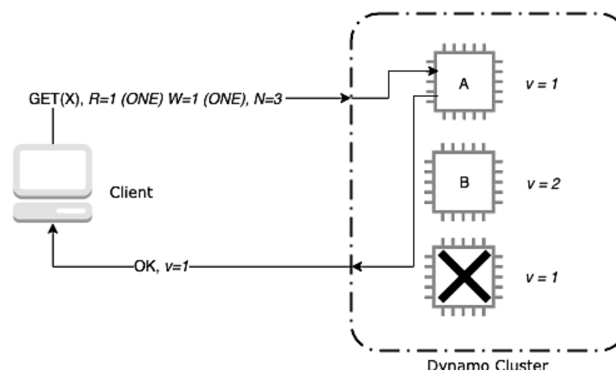


Fig. 3. Data request on abnormal operation where a node fails, and eventual read consistency is set (i.e. *ONE*).

For Dynamo-based databases, high availability does not necessarily ensures write persistence. When addressing the concept of availability, in terms of service availability and not data availability, i.e. when a request is made, a successful response is given even if such key does not exist anymore. The response is successful, even if the response refers to the inexistent of such resource. There may be the case when, for instance, a configuration of *W=1 (ONE)* is set and the same node crashes right afterward, the data is lost and there is no acknowledgment to the client that such abnormality had happened. To avoid such events, *W values* greater than 1 increase data redundancy and, consequently, the probability of all replicas that contain the data fail is diminished.

5 CBench-Dynamo

A benchmark is a standardized tool to evaluate and compare competing systems or components according to specific characteristics. These characteristics can be performance, dependability, among others.

According to [5] the benchmarks can be categorized into three types: specification-based benchmarks, kit-based benchmarks, and a hybrid based on the latest two. Specification-based benchmarks are simulated based on a specific business problem by imposing certain functions that must be achieved, such as required input parameters and expected outcomes. This type of benchmark imposes a big development investment on presenting multiple implementations for the same problem and proceed with an evaluation of that set of development. While for specification-based benchmark, the specification is a set of rules implemented by the third party to load and run the benchmark. The Kit-based benchmarks use the specification as a guide for implementing the benchmark kit. A hybrid category can be provided mostly as a kit but allows some functions to be implemented depending on each individual benchmark run.

In this section, we propose CBench-Dynamo, a consistency benchmark that is a standard procedure to evaluate and compare consistency in the System Under Test (SUT). The specification we aim to present proposes a benchmark approach to test consistency and availability in Dynamo-based NoSQL databases while subjecting these systems to network partition events. Therefore, we aim to contribute towards standardizing consistency benchmarking and lead vendors to better understand which system better suits their requirements.

5.1 CBench-Dynamo Properties

Benchmark researching and industry participants describe a benchmark into the following properties [5]: Relevance, Reproducibility, Fairness, Verifiability, and Usability.

Although the proposed benchmark, CBench-Dynamo can be adapted to run in dedicated instances it has only been tested with Amazon EC2 instances. Some orchestration playbooks, such as easy instance setup, must be adapted to work on dedicated machines. However, we aim to make the system more generic and versatile in future work. For the present paper, our goal is to define the workload approach and present preliminary empirical results from the benchmark tests using this workload.

Relevance. Relevance is the most important property when defining a benchmark [5]. The relevance of a benchmark splits up into two dimensions, the spectrum of its applicability and the degree of relevance in the given area.

The CBench-Dynamo is designed to target all Dynamo-based databases and the area of relevance is the study of the properties consistency, availability and network partition tolerance, of a horizontal-distributed database system. This benchmark aims to be a framework to facilitate the decision process of choosing the most appropriate NoSQL database depending on the degree of performance, availability, consistency, and network fault tolerance required for running a given application.

Reproducibility. Reproducibility will be attained as CBench-Dynamo exports the instances' hardware and software facts via Ansible. In addition, the workload specs will be also exported at the end of the associated run.

The goal of this extensive and detailed description is for other people to obtain identical results by configuring the whole system as described.

Fairness. Fairness is the ability of the results being supported by the system merit without artificial constraints. To reach fairness a set of artificial constraints must be consent and well defined. CBench-Dynamo defines the following constraints:

- The SUT must be a Dynamo-based NoSQL database system, e.g. Cassandra;
- The SUT must have the same hardware, network and operating system components when comparing benchmark test results targeting similar SUT;
- The Workload Coordinator must support a JVM to run the benchmark test and Python to analyze and translate the data into meaningful measurements;
- The fact that the Workload Coordinator uses Java and Python to coordinate the benchmark and post-process all the data, respectively, makes the system highly portable and therefore fair as JVM and Python based applications can run virtually in any system.

Verifiability. It is important that results are trustworthy. Results must be validated and decrease the possibility of chance or manipulation.

CBench-Dynamo results from academic work, all the workloads presented here were subject to peer-review by other researchers.

Usability. Usability is the degree of how easy a system is to use. The CBench-Dynamo has detailed instructions for making the benchmark easy to use and several layers of abstractions were taken into account so that only a minimum input is needed to start a benchmark test.

All the proposed benchmark modules are hosted on GitHub. There is a repository for each of these modules, modified YCSB [13], analyzer [14], and orchestration playbooks [15].

5.2 Architecture Specification

CBench-Dynamo is composed of a **Workload Coordinator**, a **Load Balancer** and a Dynamo cluster (**SUT**). All these components are orchestrated by the **orchestrator** via Ansible playbooks, as illustrated in Fig. 4.

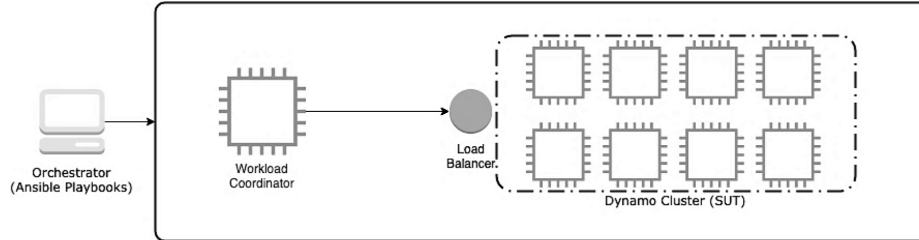


Fig. 4. CBench-Dynamo Architecture Specification.

Orchestrator. The *orchestrator* via Ansible playbooks allows the vendor to configure and run a benchmark test via the *Workload Coordinator*. The following configurations are needed prior a benchmark testing:

- IP addresses of the instances of the SUT;
- IP address of the load balancer that serves as the external gateway of the SUT;
- YCSB project home directory;
- Analyzer project home directory;
- Analysis results output directory;
- Ansible facts output directory;
- List of workload configurations with the following parameters: {*workload description, YCSB database driver descriptor (e.g. cassandra-cql), write and read consistency levels (i.e. ONE, QUORUM or ALL), number of threads used, number of objects to update, number of updates/versions per object*}.

The *orchestrator* is responsible for the following benchmark testing stages:

- Setup a dynamo cluster composed of *AWS EC2 Ubuntu 16.04 Xenial 64 bits* clean instances (only Cassandra setup was implemented);
- Prepare the data space for running the customized YCSB workload, i.e. a table called *workload* with two string-type fields, *y_id* and *version*;
- Request the Workload Coordinator to run a list of given workload configurations and reboot the SUT in each iteration;
- Request the Workload Coordinator to analyze the test output it has collected at the end of the test;
- Download from the *workload coordinator* the analysis output as a .csv file containing all the measures. Each line is the result of a single workload test.

Workload coordinator. The workload coordinator is responsible for running CBench-Dynamo workloads.

Load Balancer. The load balancer is responsible for uniformly distribute the query load throughout the SUT.

SUT. The SUT is composed of a dynamo-based cluster.

5.3 Workload

This paper proposes a workload that evaluates how consistency, performance, and availability are affected when consistency is configured either to prefer a high-available system or a high-consistent system while in a distributed system, such as Dynamo-based databases, where network partition events may occur.

The proposed workload is a customized YCSB workload and follows the methodology proposed by Bermback and Tai [8]. Bermback and Tai propose a benchmark methodology to study how Amazon S3 handles consistency over a long time period. This long-term experiment proposes a single writer and a variable number of readers as Fig. 5 suggests. To achieve a uniformly load throughout the cluster's replicas and avoid always hitting the same replica, writer and readers interact with the cluster through a load balancer.

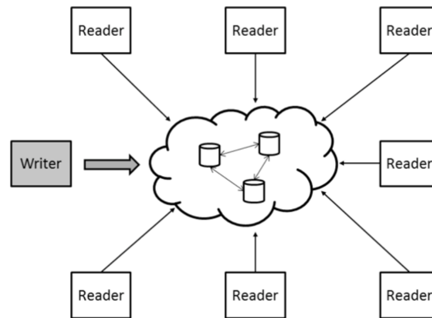


Fig. 5. Bermback and Tai's long-term benchmark approach.

Our benchmark is composed of two stages, the load, and run stages. The load stage is off the record for benchmark purposes. This stage's goal is to load all the objects into the database. These objects are composed only of two fields, *key,* and *version*.

During the benchmark run phase, both update and read operations occur uniformly. When configuring a workload run, the parameters *threads* indicate how much writers and readers will be running, as only one writer is used, the number of readers is calculated as *threads-1.* Each write operation increments a given object's version and each read operation reads the version of a given object.

The writer and the readers each have their task plan pre-generated at the beginning of the test and the benchmark ends when all the objects have been updated and read from until the pre-configured final version. Each operation is registered into a common file and follows the structure represented in Fig. 6.

...
writer_id:*0,* ***key****:9345f1bae61442dab3f167c02d19a4a8,*
timestamp:*17309459474301,* ***version:****1*
...
reader_id:*2,* ***key:****9345f1bae61442dab3f167c02d19a4a8,*
timestamp:*17309253174394,* ***version:****0*
reader_id:*1,* ***key:****9345f1bae61442dab3f167c02d19a4a8,*
timestamp:*17309253174398,* ***version:****UNAVAILABLE*

...

Fig. 6. Proposed benchmark's results data structure.

The generated data is sufficient to infer whether a consistency anomaly had happened. For a given object's key, if a read operation returns a version inferior to a version already written by some write operation in the past, there was a consistency anomaly.

At the same time this occurs, there is a module that is disconnecting from time to time one instance at a time from the cluster to simulate network partition events (see Fig. 7). For every operation that the cluster was not able to retrieve a successful answer, the version assumes the *UNAVAILABLE* value as Fig. 6 suggests.

All consistency anomalies are then processed and translated into the following measurements: *availability probability*, *consistency probability*, *write latency*, and *read latency*.

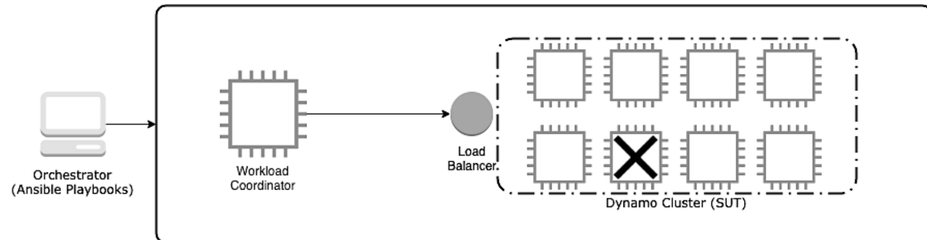


Fig. 7. Network partition event scenario.

6 Benchmark Testing

For our first test of the proposed benchmark, we chose Cassandra as our SUT. Cassandra is a database system built with distributed systems in mind, like almost every NoSQL systems out there. Following the CAP theorem, Cassandra by default is on the AP (Availability and Network Partition Tolerance) side, hence prioritizing high-availability when subject to network partitioning. As we will further see, Cassandra's consistency can be tuned to be a CP (Consistency and Network Partition Tolerance) database system, so it becomes a strong consistent database when subject to network partitioning [12].

6.1 Testing Architecture

CBench-Dynamo requires an architecture composed of an *orchestrator*, a *workload coordinator*, and a *dynamo cluster* as the SUT. The following architecture was defined for our first test (see Fig. 8):

Orchestrator. The orchestrator is a MacBook Pro 13-inch, 2017, 2.3GHz Intel Core i5 with 8GB of RAM.

Workload Coordinator. The workload coordinator is an Amazon EC2 C5n.xlarge instance (4 vCPUs, 10.5GB RAM).

SUT. The System Under Test is a Cassandra cluster composed of eight Amazon EC2 M5d.large instances. Each cluster instance will be rebooted and reloaded between workloads by the Orchestrator and the Workload Coordinator.

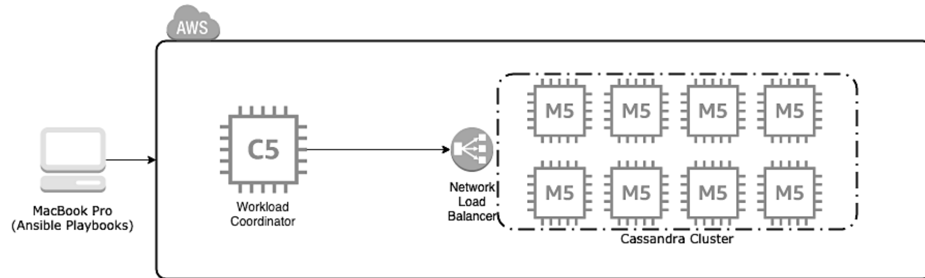


Fig. 8. Testing architecture.

6.2 Experiment

In this section, we present the results obtained after running the proposed workload configurations. Our experiment targets an 8-node Cassandra cluster and combines different consistency configurations, i.e. *ONE, QUORUM, ALL*, as described in Table 1. The common input parameters for every configuration are the following:

- Replication Factor: 3;
- Total number of objects: 1.000.000;
- Versions/updates per object: 2;
- Network partition event duration: 2s;
- Interval between network partition events: random between [1s, 25s].

Table 1. Cassandra's benchmark workload configurations.

Configuration	SUT	Write Consistency	Read Consistency
1	Cassandra	*ALL*	*ALL*
2	Cassandra	*ONE*	*ONE*
3	Cassandra	*QUORUM*	*ONE*
4	Cassandra	*ONE*	*QUORUM*
5	Cassandra	*QUORUM*	*QUORUM*

Consistency and Availability. When in a configuration where both read and write consistency is ALL we achieve results of Strong Consistency while compromising availability. This happens as theorized because all replicas must be involved before returning to the client. If some replica is down, resulting from a network partition event, the request can't fulfill and the response to the client reports an unavailable service. Although this configuration generated an availability of 99.7539%, the industry does not consider this value high. Availability is usually represented by how many nines the availability probability has (see Table 2). The value we attained in the *ALL-ALL* configuration only has 2-nines, which means that this number only falls into the second level of availability, hence translating into 3.65 days of availability when rounding down the number to 99.0000%. Many businesses that require high-availability may fail with such a long unavailable service time.

Table 2. Availability and nines notation [16].

Availability (%)	Downtime per year
90.0000 (one nine availability)	36.53 days
99.0000 (two nines availability)	3.65 days
99.9000 (three nines availability)	8.77 hours
99.9900 (four nines availability)	52.60 minutes
99.9990 (five nines availability)	5.26 minutes
99.9999 (six nines availability)	31.56 seconds

In the other hand, when querying Cassandra with no consistency constraints by setting both read and write operations to involve just one replica (write consistency = ONE and read consistency = ONE), we achieved 100% of availability, but we have compromised consistency down to the lowest value achieved in the whole experience.

As of configurations using QUORUM combined with ONE, we achieved a more balanced consistency/availability relation. As we had chosen a replica factor of three, the *QUORUM* involves two replicas when processing a client request. When reading with *ONE* and writing with *QUORUM*, the request may involve the third replica that was not part of the QUORUM for that given data object, hence returning an outdated version. When inverting the order, the *ONE* in the writing and the *QUORUM* in the reading, it seems not to have such a drastic decline in consistency, however availability loses a nine.

For a *QUORUM-QUORUM* configuration, we achieved strong-consistency and high-availability. This configuration can tolerate some network partition events unless the number of replicas down compromises the quorum. Because our network partition events had disconnected one replica at a time, the quorum had never been compromised, hence the results we had and represented in Fig. 9.

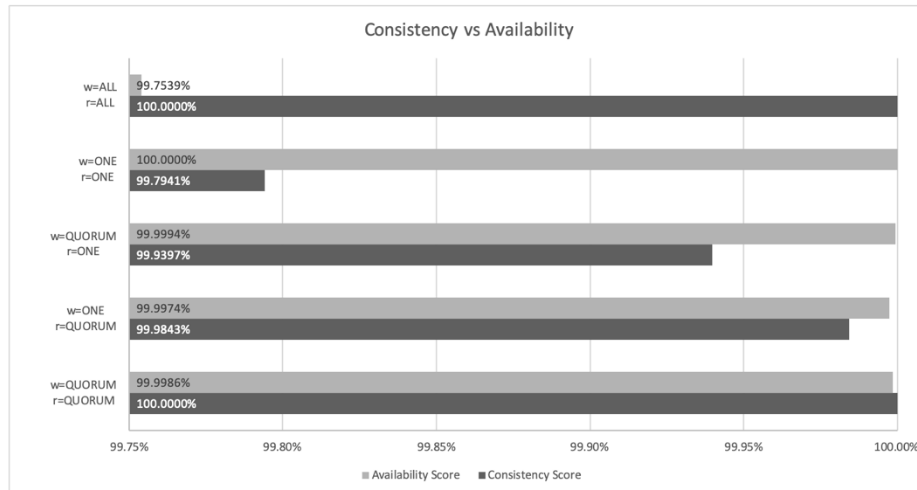


Fig. 9. Consistency and availability results for all configurations.

Performance. Our second analysis is in terms of read and write operation latencies given a consistency setting. As Fig. 10 illustrates, for an *ALL-ALL* configuration we achieved as expected the highest latencies of all configurations because all replicas had to be involved in read and write operations.

For the *ONE-ONE* configuration, because only one replica needed to be involved in read and write operations, the latencies are the lowest among all configurations tested when combining the two latencies. However, when compared solely on mixed *ONE-QUORUM* configurations, *ONE* latency in these last configurations are better.

Finally, for *QUORUM-QUORUM* configuration we achieved the most balanced configuration between latency, consistency, and availability.

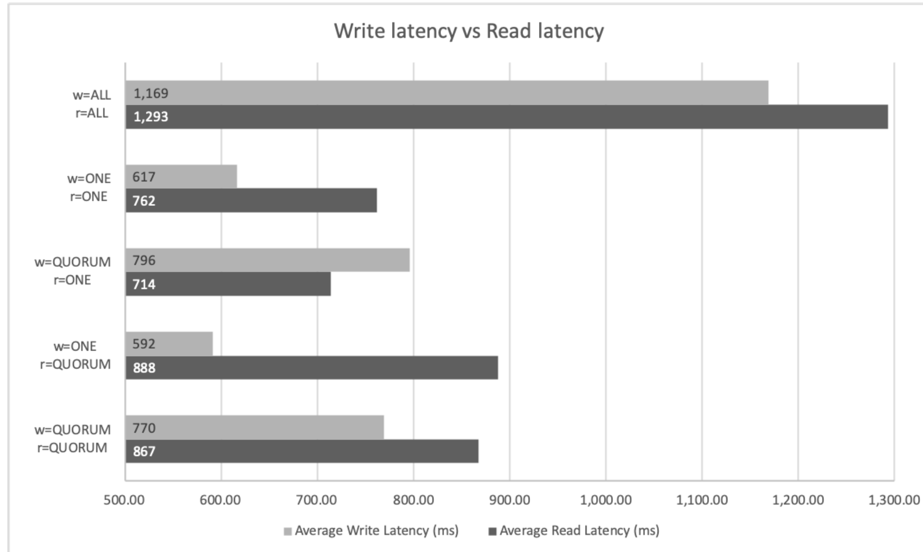


Fig. 10. Write latency and read latency results for all configurations.

7 Conclusions and Future Work

In this paper, we have proposed CBench-Dynamo as a new benchmark methodology focused on study the three properties of the CAP theorem, consistency, availability, and network partition tolerance. From the best of our knowledge, this benchmark specification for studying performance, consistency, and availability on different consistency configurations while subjecting the SUT to network partition events has never been proposed before.

In this paper, we have conceptualized, defined, and experimented our proposed benchmark resulting in interesting data on how consistency and network partition events influence consistency, availability, and performance. This benchmark is a valuable tool for testing already existing NoSQL databases against the client requirements, but also to test new databases implementations that aim a certain level of performance, consistency, availability, and network partition tolerance. We also made CBench-Dynamo benchmark available on GitHub [14-15].

For future work, we intend to support more NoSQL databases and evolve the proposed benchmark to be a well industry establish framework to test NoSQL databases so that the users have a deeper understanding of the tradeoffs of each configuration and what system and system configurations suits better their requirements.

References

1. Brewer E (2000) Towards Robust Distributed Systems. Pod. (Principles Distrib. Comput. Conf. 1–12
2. Gilbert S, Lynch N (2002) Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News 33:51–59
3. Pritchett D (2008) BASE: An Acid Alternative. Queue 6:48–55
4. Yahoo! GitHub Repository: YCSB (Yahoo! Cloud Serving Benchmark). https://github.com/brianfrankcooper/YCSB
5. Henning JL, Arnold JA, Lange K-D, et al (2015) How to Build a Benchmark. 333–336
6. Cooper BF, Silberstein A, Tam E, et al (2010) Benchmarking Cloud Serving Systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. ACM, New York, NY, USA, pp 143–154
7. Bailis P, Venkataraman S, Franklin MJ, et al (2014) Quantifying Eventual Consistency with PBS. Commun ACM 57:93–102
8. Bermbach D, Tai S (2014) Benchmarking eventual consistency: Lessons learned from long-term experimental studies. Proc - 2014 IEEE Int Conf Cloud Eng IC2E 2014 47–56
9. Patil S, Polte M, Ren K, et al (2011) YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing. ACM, New York, NY, USA, pp 9:1--9:14
10. DeCandia G, Hastorun D, Jampani M, et al (2007) Dynamo: Amazon's Highly Available Key-value Store. Proc Symp Oper Syst Princ 205–220
11. Bailis P, Venkataraman S, Franklin MJ, et al (2012) Probabilistically Bounded Staleness for Practical Partial Quorums. Proc VLDB Endow 5:776–787
12. Diogo M, Cabral B, Bernardino J (2019) Consistency Models of NoSQL Databases. Futur Internet 11:
13. Diogo M (2019) GitHub Repository: YCSB-Consistency
14. Diogo M (2019) GitHub Repository: CBench-Analyser. https://github.com/migueldiogo/cbench-analyser
15. Diogo M (2019) GitHub Repository: ansible-dynamo-clusters. https://github.com/migueldiogo/ansible-dynamo-clusters
16. Marcus E (2003) The myth of the nines. In: Blueprints High Availab. https://searchstorage.techtarget.com/tip/The-myth-of-the-nines

This page is intentionally left blank.

Appendix D

TPCTC/VLDB 2019 paper presentation.

CBench-Dynamo: A Consistency Benchmark for NoSQL Database Systems

Miguel Diogo, Bruno Cabral, Jorge Bernardino

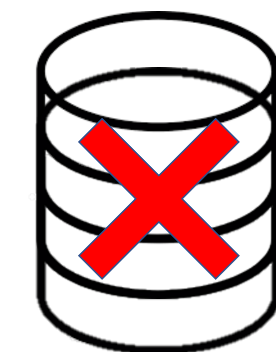
Universidade de Coimbra

Portugal

August 2019

TPCTC Technology Conference on Performance Evaluation and Benchmarking

Vertical Scaling

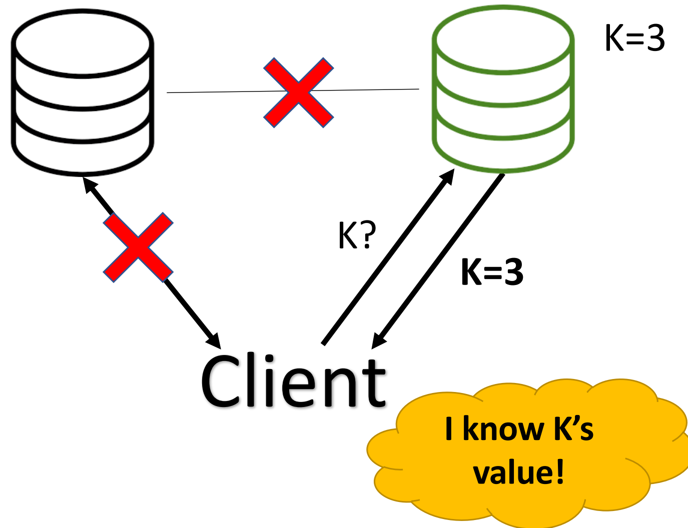


K???

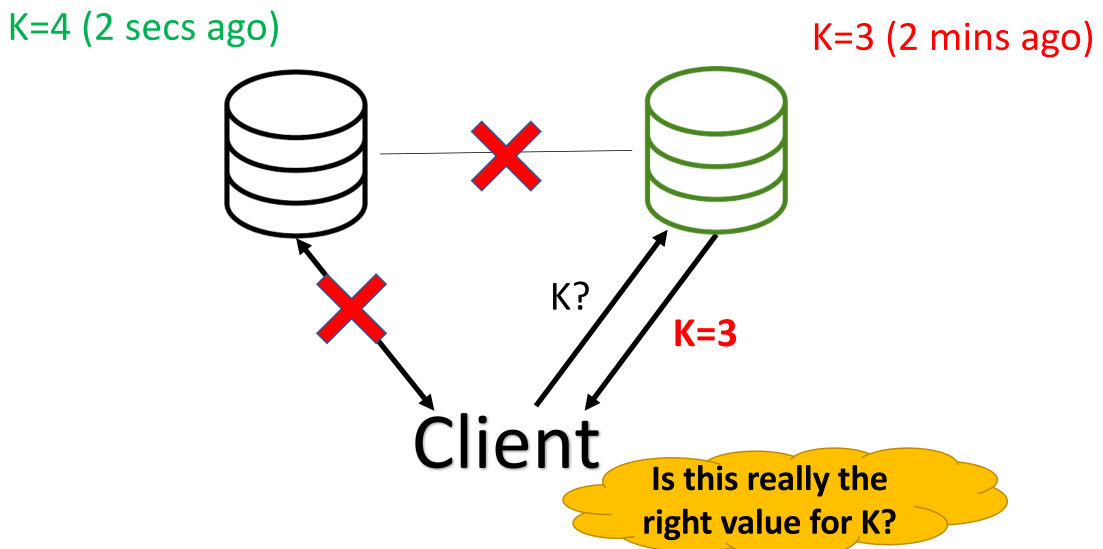
Client

I really need to know K's value…

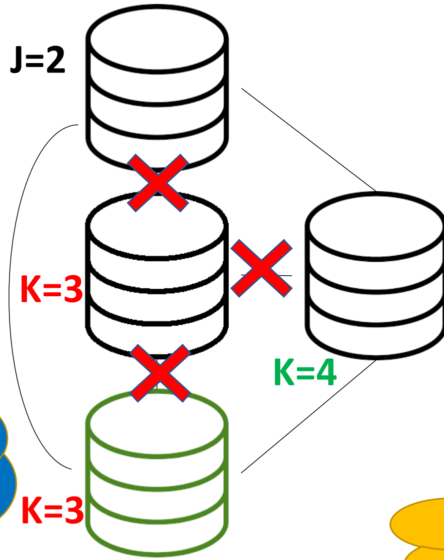
Horizontal Scaling

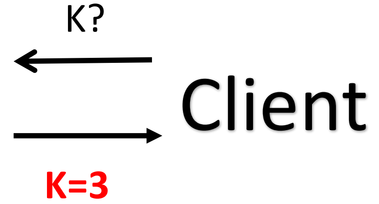


Inconsistency



Replication





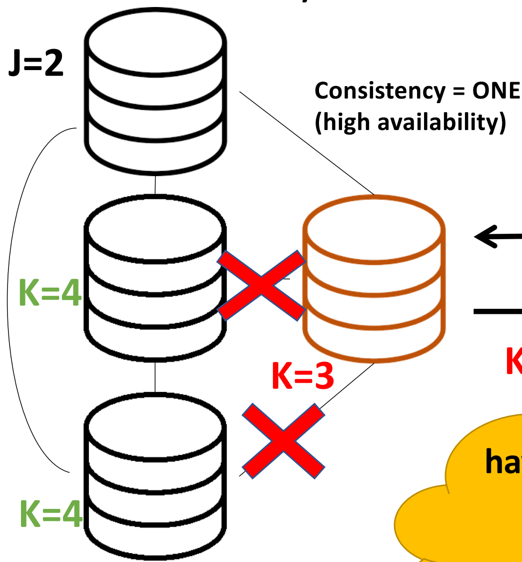
Replication Factor(N) = 3

Read Consistency ONE = 1

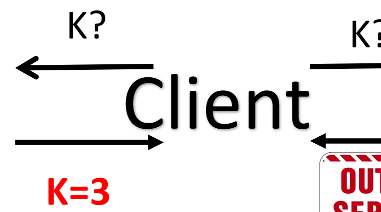
Can respond. I only need 1 node online from the 3.

I don't care if it is correct or incorrect. Eventually it'll be consistent with what I expect

Availability vs. Consistency



Consistency = ONE (high availability)



Consistency = ALL (strong consistency)

K?

OUT OF SERVICE

I lost connection with my peers. Can't decide on my own!

J=2

K=4

K=4

K=3

Do I prefer always having a response even if not consistent (availability) or a consistent response (consistency)?

Problem

- How do we correlate/evaluate consistency with other properties such as latency and availability when network partition events occur?

Solution

CBENCH-DYNAMO

- A benchmark methodology that correlates consistency, performance, and availability while network partition events are involved.

- We made a custom YCSB workload to achieve this. YCSB is a benchmark framework for NoSQL databases.

Relevance

Aims to be a framework to facilitate the decision process of choosing the most appropriate NoSQL database depending on the degree of **performance**, **availability**, **consistency**, and **network fault tolerance.**

Reproducibility

CBench-Dynamo exports the instances’ hardware and software facts via Ansible orchestration so others can reproduce this work. Also all this work is available on Github and it’s open source.

Fairness

CBench-Dynamo defines the following constraints to achieve fairness:

- The **SUT must be a Dynamo-based** NoSQL database system, e.g. Cassandra;
- The **SUT must have the same hardware, network and operating system** components when comparing benchmark test results targeting similar SUT;
- The **Workload Coordinator must support a JVM to run the benchmark test and Python to analyze** and translate the data into meaningful measurements;
- The fact that the Workload Coordinator uses Java and Python to coordinate the benchmark and post-process all the data, respectively, makes the system **highly portable** and therefore fair as **JVM** and **Python** based applications can run virtually in any system.

Verifiability

- CBench-Dynamo results from academic work, all the workloads presented here were subject to peer-review by other researchers.
- For future work we intend to improve this quality requirement and make it more automated.

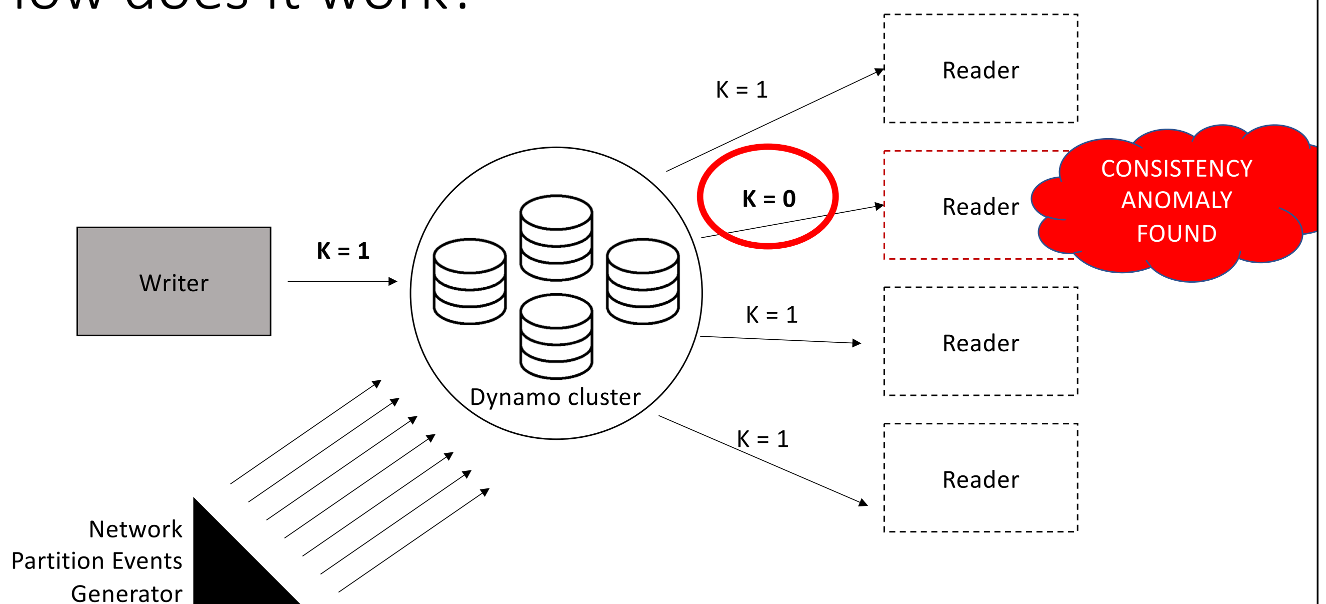
Usability

- Several layers of abstractions were taken into account so that only a minimum input is needed to start a benchmark test.

```
vars:
  - description: "{{ item.description }}"
  - db: "{{ item.db }}"
  - write_consistency: "{{ item.write_consistency }}"
  - read_consistency: "{{ item.read_consistency }}"
  - threads: "{{ item.threads }}"
  - objects_num: "{{ item.objects_num }}"
  - version_num: "{{ item.version_num }}"
  - host: "{{ groups['nodes'] | random }}"
```

```
with_items:
    - { description: cassandra, db: cassandra-cql, write_consistency: ALL, read_consistency: ALL, threads: 4, objects_num: 1000000, version_num: 2 }
    - { description: cassandra, db: cassandra-cql, write_consistency: ONE, read_consistency: ONE, threads: 4, objects_num: 1000000, version_num: 2 }
    - { description: cassandra, db: cassandra-cql, write_consistency: ONE, read_consistency: QUORUM, threads: 4, objects_num: 1000000, version_num: 2 }
    - { description: cassandra, db: cassandra-cql, write_consistency: QUORUM, read_consistency: ONE, threads: 4, objects_num: 1000000, version_num: 2 }
    - { description: cassandra, db: cassandra-cql, write_consistency: QUORUM, read_consistency: QUORUM, threads: 4, objects_num: 1000000, version_num: 3 }
```


How does it work?



Workload

The writer is responsible for incrementing key's version to a given limit (2 in this case)

OBJECTS_NUM = 6

Key	Version
8314e14ba5b24be3ae4dbdfe45401524	2
966eae76c566408396983fbb7aefdd77	2
2fa8d8086b9543548fd6a6e99bc736b1	2
3cdaf599580c48a09290a14efae689e0	1
012e9eb2ef294ac185062ea874b24f5f	1
04170cc5d7d84202aed116fc53734409	0

VERSION_NUM = 2

The benchmark ends when all keys reach the version 2

Output Data

...

writer_id:0, **key**:9345f1bae61442dab3f167c02d19a4a8, **timestamp**:17309459474301, **version**:1

reader_id:2, **key**:9345f1bae61442dab3f167c02d19a4a8, **timestamp**:17309253174394, **version**:0

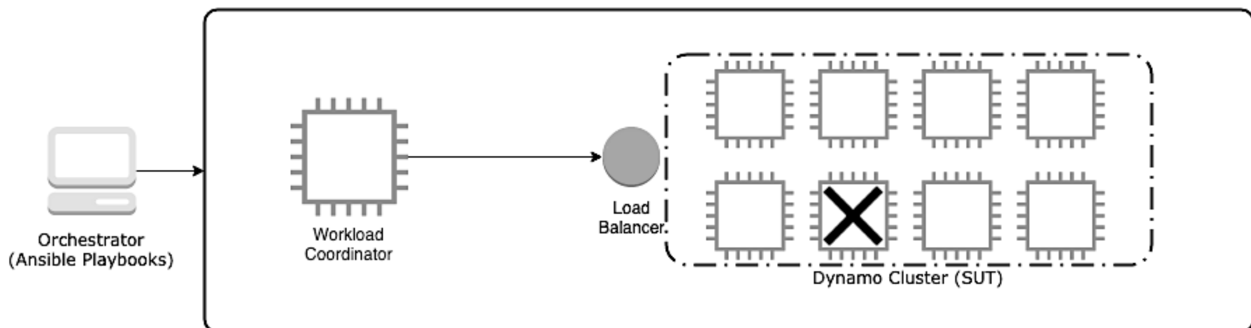
reader_id:1, **key**:9345f1bae61442dab3f167c02d19a4a8, **timestamp**:17309253174398, **version**:UNAVAILABLE

...

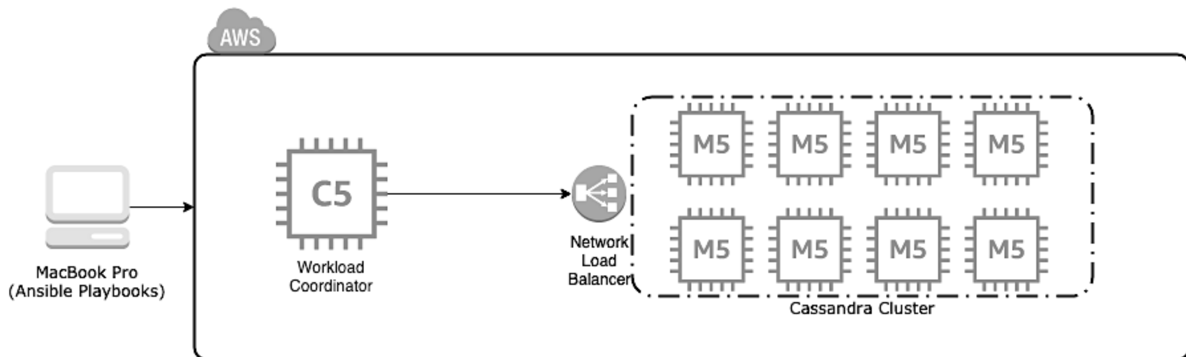
Analyzed data

average time read	average time write	Consistency score	availability score	total operations
761.87301	616.589662	99.794054	99.9999	6870716

Generic Architecture



Testing Architecture



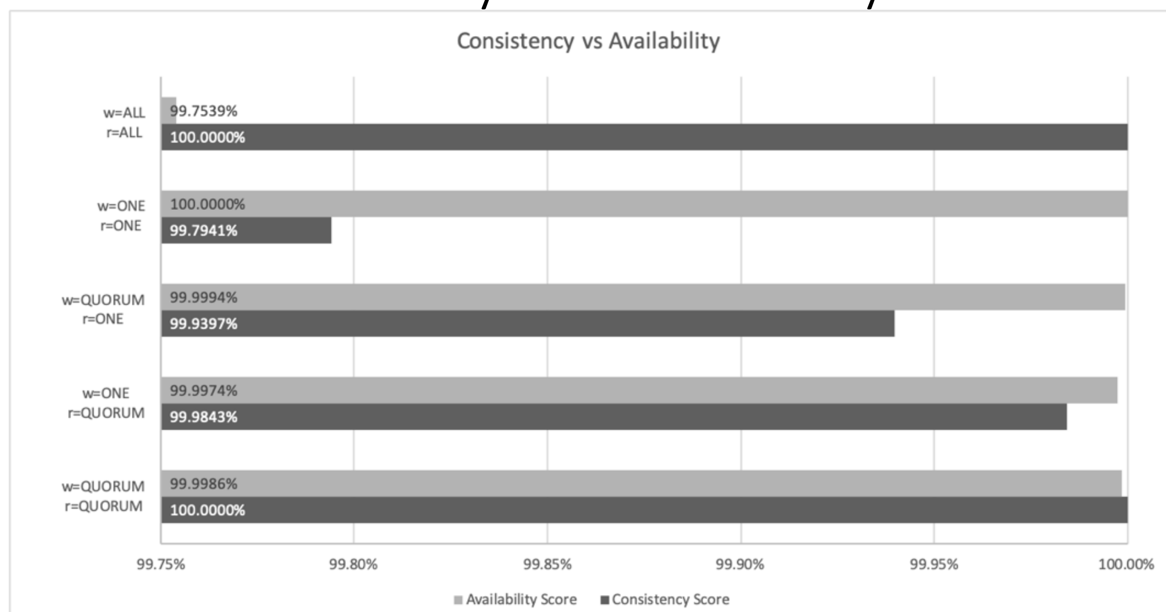
Testing configurations

Configuration	SUT	Write Consistency	Read Consistency
1	Cassandra	ALL	ALL
2	Cassandra	ONE	ONE
3	Cassandra	QUORUM	ONE
4	Cassandra	ONE	QUORUM
5	Cassandra	QUORUM	QUORUM

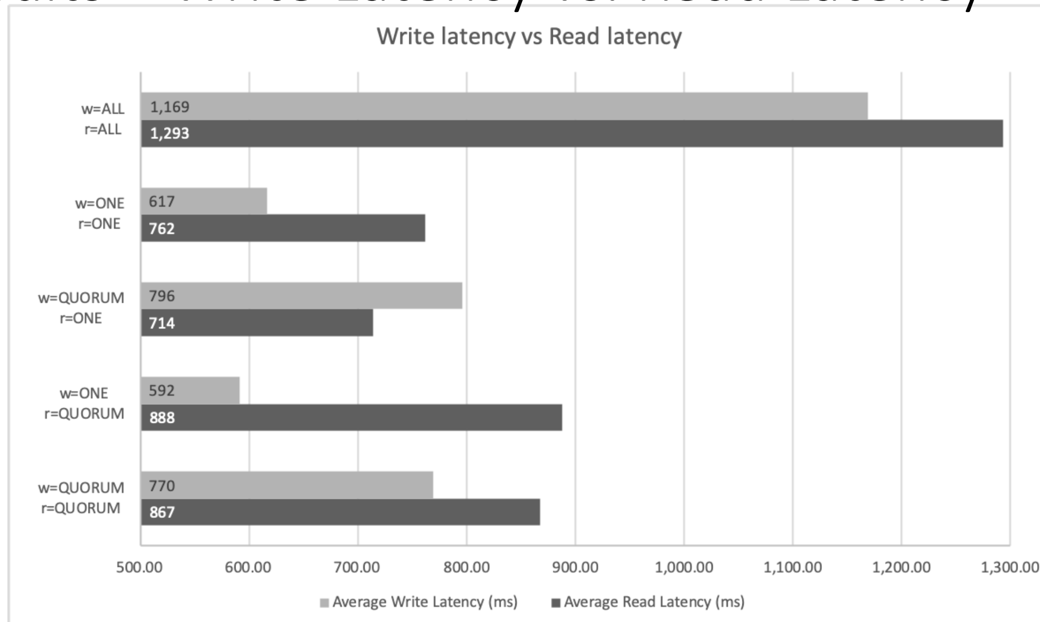
Variables Common to All Testing Configurations

- **Replication Factor:** 3;
- **Total number of objects:** 1.000.000;
- **Versions/updates per object:** 2;
- **Network partition event duration:** 2s;
- **Interval between network partition events:** random between [1s,25s].

Results – Consistency vs. Availability



Results – Write Latency vs. Read Latency



Conclusions

- Proposed CBench-Dynamo as a new benchmark methodology focused on studying the three properties of the CAP theorem, consistency, availability, and network partition tolerance.
- The first benchmark that studies consistency, availability, and performance while network partition events are involved.
- CBench-Dynamo benchmark is available on GitHub.

Future Work

- Support more NoSQL databases and evolve the proposed benchmark to be a well industry establish framework to test NoSQL databases on consistency, availability, and performance while subjecting them to network partition events.