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].

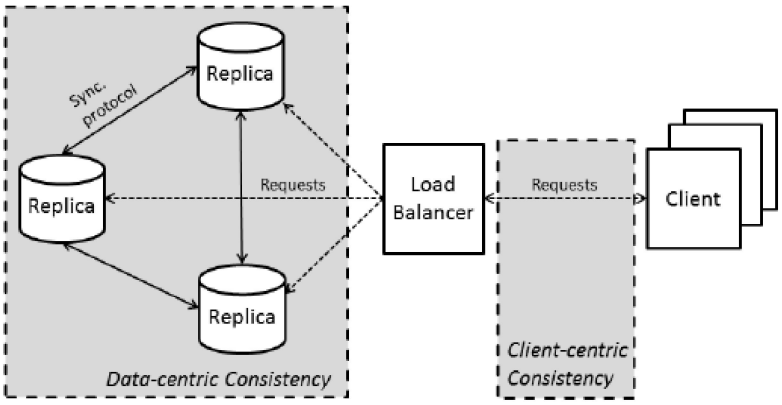

Figure 1. Data-centric and Client-centric consistencies [13].

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).

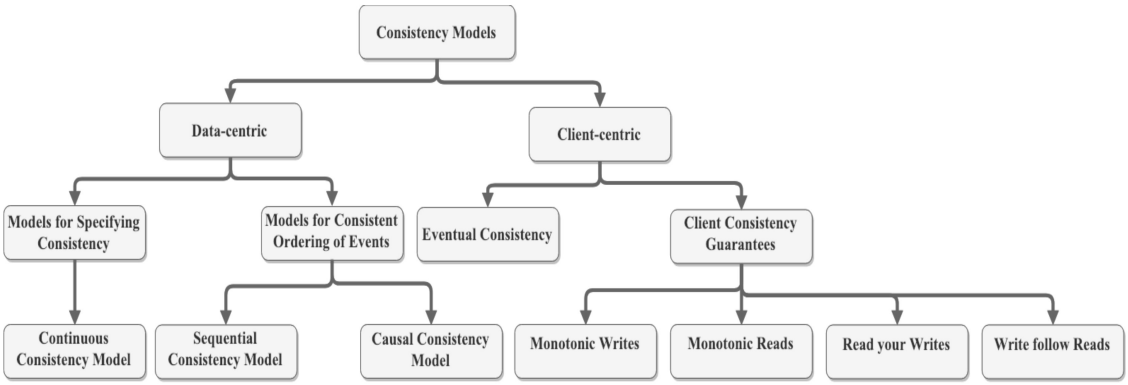

Figure 2. Consistency Models based on Reference [14].

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].

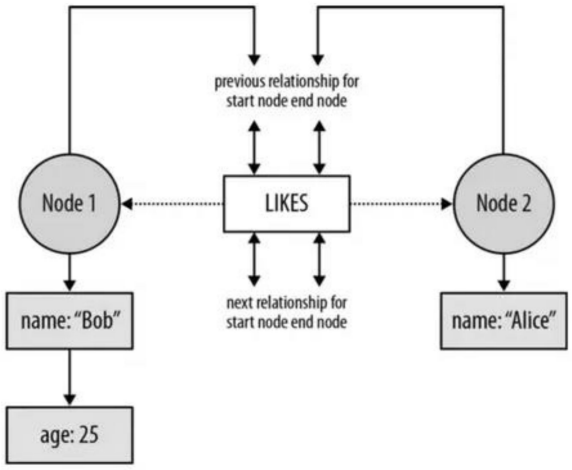

Figure 3. Neo4j Disk Data Structure. Source: Reference [32].

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:

$$\text{HASH\_SLOT} = \text{CRC16(key)} \bmod 16384$$

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.

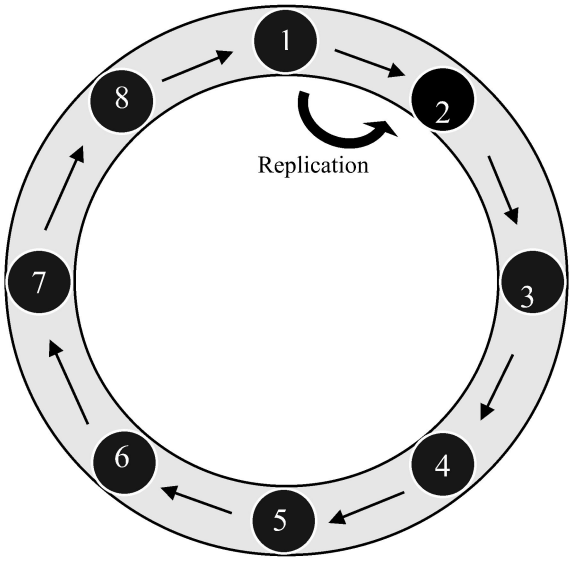

Figure 4. Cassandra Ring.

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.

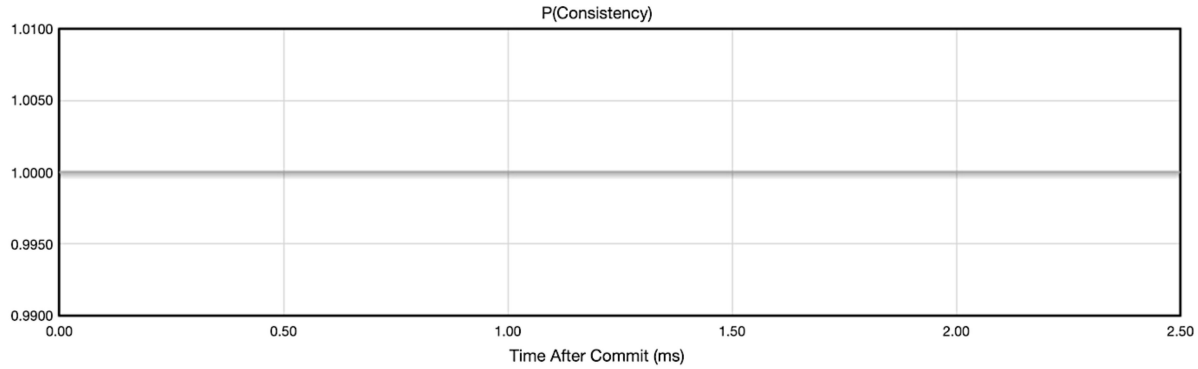

Figure 5. Probabilistically Bounded Staleness (PBS) results for (N = 5, R = 1, W = N = 5) and (N = 5, R = N = 5, W = 1).

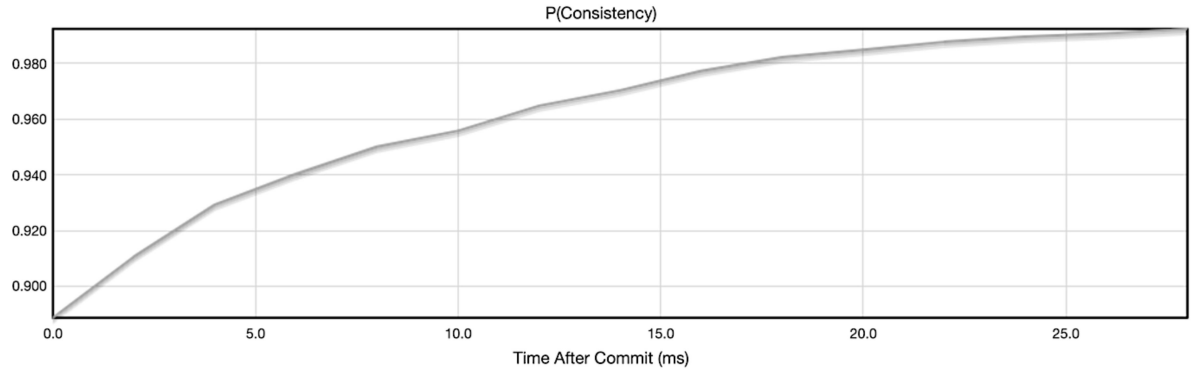

Figure 6. PBS results for (N = 5, R = 1, W = 3).

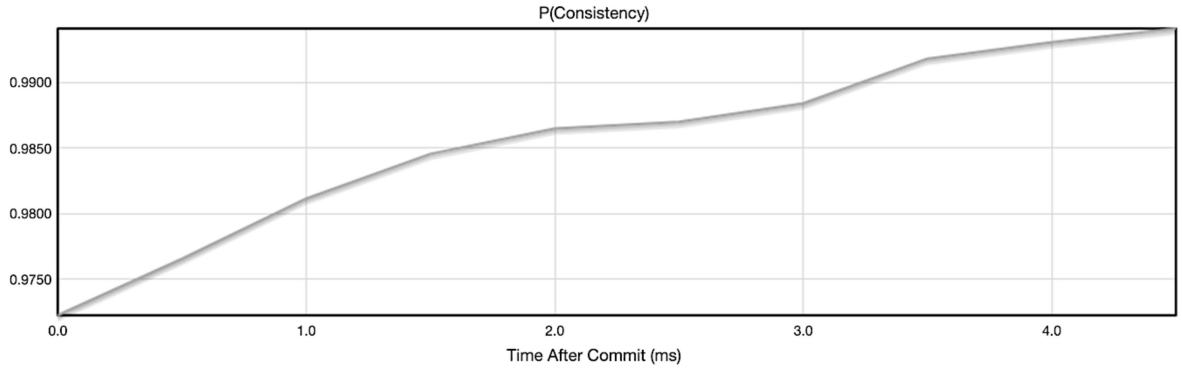

Figure 7. PBS results for (N = 5, R = 3, W = 1).

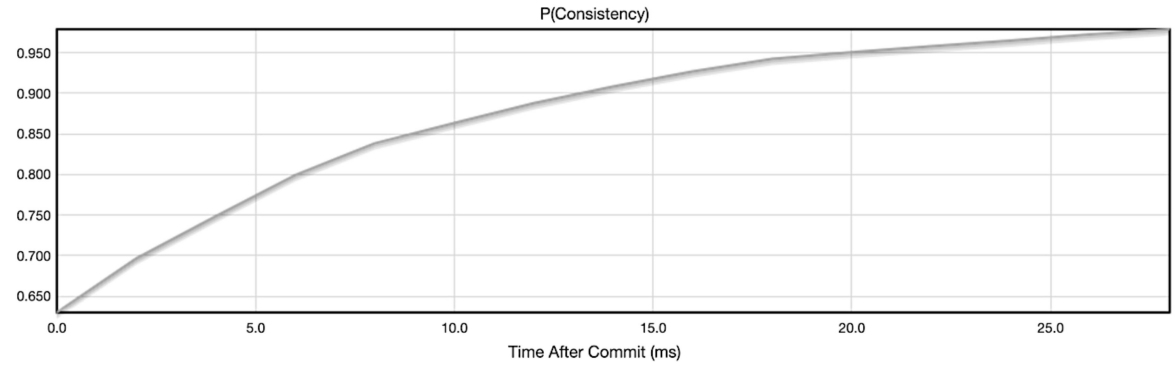

Figure 8. PBS results for (N = 5, R = 1, W = 1).

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.

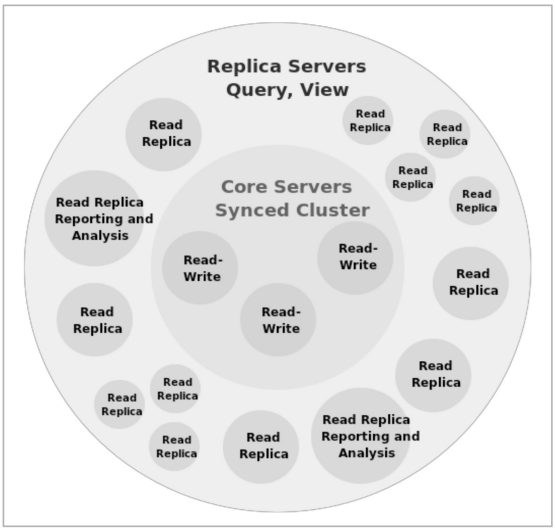

Figure 9. Neo4j Cluster Architecture.

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.

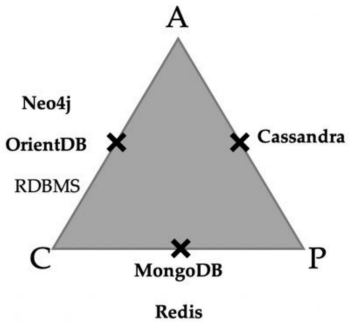

Figure 10. Consistency, Availability, and Network Partition Tolerance (CAP) Theorem and classification of databases based on their default configurations.

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]