Chapter One

Introduction

1.1 Background

In recent days the web applications and web services are uses the data it become must important and widely used, the web 3 and mobile applications are designed to support huge number of concurrent users by using the distribute the load across multiple nodes[1]. The latest updates to databases and data storage have created a variety of challenges. Specifically, classic relational databases (such as Postgres SQL, MySQL, and others) are having a difficult time keeping up with current changes because of their reliance on set schema and poor scalability [1].

Databases is the best tools appearance of computer innovation and are pivotal parts of data frameworks. Utilizing the proper databases is fundamental for viably putting away and recovering information. Structured Query Language, or SQL, is a term frequently used to refer to relational database systems. When it comes to databases, relational databases have long been the standard. They are well-liked because of their dependability and usefulness. Yet, these databases have disadvantages when it comes to non-structured and massive amounts of data in the current context of a data environment that is always evolving. As an outcome, ***NoSQL*** databases are a new kind of database.

As a result of the new technology trends in a web, social-media network platforms, mobiles, and the Internet of Things, that generate a new type of data managed by special types of application, this huge data was named a massive data applications that are used

to manage many types of data like structured-data, semi-structured-data, and unstructured-data that has arisen during in early 2000s[2]. Some applications have a number of specifications for data storage, such as: scalability, which empowers direct flexibility to the gigantic sums of information and the quickening inquiry handling speed, Tall accessibility and blame resistance to fulfill client demands indeed within the occasion of hardware/software disappointment or overhaul occasions, Corrosive qualities to allow profoundly reliable database questions, and There are four Corrosive highlights that make exceedingly reliable database inquiries simpler.

On the other hand, ***NoSQL*** databases were specifically created to meet this big scale requirements and relational databases' store capacity limits. A non-relational, distributed, and horizontally scalable database is known as ***NoSQL*** (Not Only SQL). Although data consistency is an essential quality, most ***NoSQL*** databases do not guarantee it, unlike relational databases. The data consistency attribute states that the same set of data values are consistently supplied to all instances of an application. When competing copies of the same data emerge in several locations, there is data inconsistency[3]. Information that has a rating of inconsistency is unreliable since it will be challenging to verify which version of the information is accurate or not.

1.2 Research Contributions

The main and principal contribution brought forth by this study is the provision of a tool, specifically designed for system designers, that serves the purpose of assisting them in the process of selecting the most optimal configuration settings. This selection process is aimed at ensuring the achievement of strong data consistency within *NoSQL* databases. Moreover, it is important to note that this selection is not made in isolation, but rather

takes into account the consideration of high performance metrics, such as a high level of throughput and low latency. Furthermore, in addition to this primary and fundamental contribution, this study encompasses a range of other significant and noteworthy efforts.

Firstly, a comprehensive and extensive survey has been conducted, which explores and delves into the various types of *NoSQL* databases that exist. This survey also aims to uncover and investigate the diverse consistency models that these databases employ. Additionally, it also addresses and analyzes the challenges and limitations that are associated with the task of ensuring transaction consistency across these databases.

Secondly, a model has been developed as a result of this study. This model proposes and puts forth different levels of consistency that can be implemented and observed within *NoSQL* databases. This model serves as a guide and provides a framework for system designers to follow in order to achieve the desired level of consistency within their databases. Lastly, this study has conducted a series of comprehensive tests and evaluations of the proposed model. These tests and evaluations have been carried out using multiple models of *NoSQL* databases. The purpose of these tests and evaluations is to validate and verify the effectiveness and efficiency of the proposed model in achieving the desired levels of consistency within *NoSQL* databases.

Overall, the contributions made by this study significantly add to and enhance the understanding and advancement of transaction consistency within *NoSQL* databases. These contributions serve to shed light on the various factors and considerations that need to be taken into account when designing and implementing *NoSQL* databases, with the goal of achieving strong data consistency. By considering and addressing the challenges and limitations associated with transaction consistency, system designers can make

informed decisions and implement appropriate strategies to ensure the desired level of consistency within their databases.

1.3 Research Objectives

The conduct and the enhancing performance of transaction consistency in *NoSQL* databases is a crucial area of research, as it can significantly impact the scalability, availability, and overall performance of *NoSQL*-based applications. Here are some of the key objectives of this study:

1. To understand the challenges of achieving ACID compliance in NoSQL databases.
2. To examine existing methods and tools used to improve NoSQL transaction consistency
3. To understand the trade-offs between consistency and performance
4. To propose a novel model for optimizing consistency in NoSQL transaction management.
5. To analyze the performance, scalability, and consistency guarantees of the model compared to existing solutions.

By pursuing these objectives, the researchers can gain a deeper understanding of consistency in *NoSQL* transactions, leading to improved data integrity, application reliability, and overall system performance.

1.4 Research Motivation

The trend of migrating towards *NoSQL* solutions has gained traction among computer companies and users. However, the successful implementation of transactional data

management systems in *NoSQL* databases faces a significant obstacle in achieving consistency. In the realm of big data and real-time web applications, companies like Google, Facebook, and Amazon highlight the need for strong consistency requirements in web-scale systems. Modern *NoSQL* databases, such as Cassandra, MongoDB, ScyllaDB, and DynamoDB, prioritize eventual consistency over immediate availability for client applications. This approach places a burden on developers, as they must invest considerable effort in creating distributed applications for *NoSQL* databases that adhere to eventual consistency. Furthermore, this flaw in functionality forces developers to work harder to mitigate the risk of data staleness and conflicts among multiple clients. While it may not be feasible to maintain consistency for every type of data, such as Facebook and search cache, the reliance on eventual consistency represents a significant departure from the robust guarantees offered by traditional databases, thus burdening software developers[4].

Developing advanced and scalable systems that meet the demands of poor consistency guarantees is an exceedingly challenging task. Therefore, it is crucial for application developers to reject subpar consistency and explore scalable *NoSQL* database designs that provide more reliable data consistency. This requires a thorough investigation into the causes of this contradiction. *NoSQL* databases are a relatively new venture, and their popularity has grown in recent years. Consequently, many database designers are still in the process of acquainting themselves with this technology.

Given these observations, it is clear that addressing performance optimization concerns related to this emerging technology is a worthwhile endeavor. Thus, application developers must prioritize tackling these concerns.

1.5 Research problems

NoSQL databases present several obstacles. One primary challenge lies in their absence of transactional consistency, a characteristic commonly found in conventional relational databases. Consequently, data may not exhibit uniformity across all nodes within the cluster in certain scenarios. Furthermore, *NoSQL* databases may necessitate a higher degree of expertise for setup and management compared to their relational counterparts. Additionally, the absence of a predefined schema can complicate the preservation of data quality and consistency. Nevertheless, despite these hurdles, *NoSQL* databases remain a favored option for contemporary applications due to their advantageous qualities of scalability, flexibility, and performance.

1.6 Research Questions

The questions of this study are:

Research question 1: What are the different consistency models available for *NoSQL* databases?

Data inconsistency is a common drawback encountered in *NoSQL* databases, a predicament regularly faced by both consumers and developers during the execution of real-time transactions. In the realm of cloud applications, it is not uncommon for data to be partitioned across various data centers situated in multiple physical regions, a phenomenon necessitated by a multitude of reasons such as enhancing availability through data replication and optimizing scalability by distributing the workload across different nodes. Nevertheless, within this model, the preservation and monitoring of data

consistency have emerged as indispensable elements of the system's functionality and performance.

Chapter two section six of this study will identify the conditions that lead to the states of consistency models in *NoSQL* databases.

Research question 2: What are the trade-offs between different consistency models?

There are a few technologies and approaches available to preserve the consistency of *NoSQL* databases, such as Eventual, BASE, Quorum, and CAP [7] these consistency techniques were developed in response to application requirements and/or capabilities.

The level of performance of the consistency performances of two types of consistency will be proved in chapter four of the investigation.

Chapter four section 2 in this study will present the consistency model that offer the best consistency in *NoSQL* databases.

Research question 3: How can improve the performance for getting strong consistency guarantees in *NoSQL* transaction?

One of the biggest challenges with consistency in *NoSQL* databases is the trade-off between consistency and performance. Strong consistency guarantees can lead to performance bottlenecks, especially in highly concurrent environments. Weaker consistency guarantees can improve performance, but they can also lead to inconsistencies between nodes. Another challenge is that consistency approaches for ***NoSQL*** databases are still evolving. There is no single consistency approach that is ideal for all applications. Developers need to carefully consider the needs of their application

when choosing a consistency approach. Despite these challenges, optimistic about the future of consistency in ***NoSQL*** databases. The researcher are actively developing a novel consistency model that aim to improve the trade-off between consistency and performance. I am also hopeful that *NoSQL* database vendors will continue to improve their support for different consistency levels in chapter six present the proposed model of *NoSQL* Transactions consistency (PMC) architecture and details.

Chapter four section 3 present the model design and how can implement in NoSQL databases for improve the performance and getting strong consistency.

Research Question 4: Is the transaction's consistency configurations can effect on the performance of the *NoSQL* database?

It is important to choose a consistency setting that meets the needs of the application. For applications that require high consistency, it may be necessary to sacrifice some performance. For applications that require high performance, it may be necessary to accept some inconsistency. Strong consistency guarantees require that all nodes in the database have the same data at all times. This can be achieved by using locks or other synchronization mechanisms. However, these mechanisms can add overhead to reads and writes, which can reduce performance. Weaker consistency policies, such as eventual consistency, allow some nodes to have stale data for a period of time. This can improve performance by reducing the need for synchronization. However, it can also lead to inconsistencies between nodes, which can cause problems for applications that require strong consistency.

This study was test some configuration models to conduct a hybrid consistency settings can strike a balance between performance and consistency, a hybrid settings might require that all nodes have the same data within a certain time frame.

Research Question 5: Is the proposed model should address the limitations of existing approaches and offer scalability, and performance optimization.

The Researcher in chapter five in this study was developed a novel transaction's consistency model that aim to improve the trade-off between consistency and performance.

1.6 Research outlines

This thesis contained a number of chapters that can be summarized as follows

The **first chapter** contained a group of paragraphs including the introduction, objectives, presentation and description of the problem, etc.

The **second chapter** contained a set of concepts related to the problem and the theoretical framework along with the literature of the study of *NoSQL* models and reviews relevant literature in light of our research goals. Also the chapter contained a group of previous studies related to consistency in transactions in *NoSQL* databases.

The **third chapter** of this study included the methodology and conceptual designing for the model.

The **forth chapter** present the implementation of the study and its experiments, where the researcher conducted a basic experiment and testing experiments for the proposed model. And test a performance evaluation experiment setups and results was presented.

Chapter Five contained a set of results obtained by the researcher from experiments on the proposed model to solve the problem of transaction consistency in *NoSQL* databases.

Chapter sex contained the discussion of the experiments result

Finally **chapter seven** presents a summary and conclusion of the study and a set of recommendations for future studies on scientific topics related to the study.

Chapter Two

Literature Review

2.1 Background

To answer the given research question, it is necessary to concentrate on the areas listed below in order to investigate the scope of a *NoSQL* databases. This chapter discusses the characteristics of ***NoSQL*** models and presents the literature on transactional approaches for such models. The discussion provided is helps us understand the cost implications of implementing ***NoSQL*** consistency with existing approaches. They also pave the way for recent research showing how they improve recommendation solutions to two closely related problems of transactions and consistency in ***NoSQL*** database design. Although the primary focus of this chapter is the study of alternative ***NoSQL*** transactional approaches.

2.2 *NoSQL* Models

The best description of a data model is "a data model is a design model that describes just data and its connections, it contains entities," although this only refers to the data they carry, not how they own what it uses or what their job is. There are many more definitions of data models as well [5, 6] A data model is described by others as an abstract model that arranges and normalizes data items and describes how they relate to one another and the characteristics of real-world entities[7]. A data model is an overview of the representation and manipulation of real-world things and their interactions, according to A. *Silberschatz et al.*[5]. Key-value stores, wide-column stores, document stores, and object stores are the four basic forms of ***NoSQL*** storage. Each of the aforementioned

models has distinctive characteristics that make an associated store appropriate for a particular use case.

2.2.1 Key-Value Data Model

The simplest and most constantly used in ***NoSQL*** data stores are key-value stores, which handle and represent data as pairs of keys and values. The key element might be straightforward (similar to a URI, hash, or filename) or structured using compound keys. Also, it could be system or operation generated. The customer operation is in charge of handling the serialization and deserialization of the value part's data, which can be of any kind, structure, or size and is decoded as a byte array, for illustration, as a BLOB. The client operation must apply its indexing and querying because to the stored values in a schema-less form. Key-value models are thus only applicable for operations that only need a single key to recover data, similar online shopping carts, profiles, configurations, and web session information. This simple data structure enables quick data partitioning and effective data querying, both of which contribute to the high scalability of key-value. In reality, radical key-value stores include further features like indexing and querying the content of values of certain data types since numerous operations demand a value-grounded lookup of data. For case, the list data type is supported by Redis. They enable carrying out insignificant operations on list values, similar to pushing into a list without fully altering the item. Riak crucial- value3 also supports JSON and other document formats [1]. The distinction between key-value stores and other types of ***NoSQL*** stores has come blurred as a result of these added features. For illustration, a document storehouse might be compared to Riaks' key-value .Simple key-grounded query operations like GET (key), PUT (key, value), and cancel are offered by a standard key-

value database (key). The function GET (key) returns the value (or a collection of values with colorful performances) connected to the key. Put adds the crucial and value.

The simplest and most popular ***NoSQL*** stores are key-value stores, which manage and portray data as key-value pairs with the value component being particularly recognizable by an indexed key part. The value part's contents, which can be of any sort, structure, or size and is encoded as a byte array, must be serialized and deserialized by the client program. This straightforward data format allows for efficient data searching and easy data splitting. As many applications require a value-based search of data, modern key-value stores now incorporate additional functionality like indexing and querying the content of values of certain data types. In addition to JSON, Riak key-value3 supports several document types. A provides straightforward key-based query operations including GET (key), Put (key, value), and delete. If the key is not present, then only pair with the store. A fresh iteration of the stored value is added if not. Remember that updating a stored value in its whole follows modifying any individual components. Removes both the key and the value it refers to with Delete (key) (s). The details of the aforementioned operations depend on some factors, such as the consistency model, indexing, etc. These single-key operations cannot modify several values at once. These tasks can be completed quickly using the Lucene [8] or REST [9] interfaces. Figure 2.2(a) shows a straightforward Key-value store that is used by a health information management system, supposing that patients' medical records are often accessed using their SSNs and that patient data is rarely updated.

2.2.2 Column-Family Data Model

This type of data stores were improved Key-value stores with a nested Key-value pair table for the value component and a schema that is changeable. column data-store was used a table as a collection of rows, each of which reflects a highly ordered construct and is uniquely recognized by a row key and a certain number of column families. Any number of logically linked columns or cells make up a column family, which is frequently accessed as a whole for querying reasons. This explains why data is physically stored in wide-column tables as column families rather than rows. The extensible schema of a column family may be set aside updated by dynamically adding or removing columns. Every column has a name, a simple (number or text) value, or a more complicated structure like a group of columns. A value is retrieved using a triple in wide-column stores, which usually offer the storing of an arbitrary number of copies of each cell value, indexed by timestamps. The client application may assign a timestamp implicitly or the store may explicitly assign one. Wide-column stores have more complete client interfaces than Key-Value stores since their indexing and querying capabilities are based on a number data structures, including rows, column families, and columns. Figure 2.2 shows an example of how Facebook created a wide-column table for its Inbox Search feature (b). This feature allows users to search their sent and received messages using either a keyword (known as a term enquiry) or the sender or recipient's name (called interaction query) More specifically, the row-key for both term and interaction queries is the user-ID. Sender-recipient and Sent-received are two different column families that, respectively, satisfy the requirements for interaction and keyword searches. In Sent-received, the user's messages' keywords are transformed into nested column families, sometimes referred to as column families. Each column family converts the unique

message-IDs (or links to messages) into columns to reduce recurrence. Similar to this, column families are also established in Sender-Recipient using all of the user-IDs of the users that send and receive messages.

Wide-column storage can effectively divide data into vertical and horizontal rows and columns, making it suitable for storing large datasets. You should be aware that a number of wide-column stores, like Google BigTable[10], Apache HBase[11],, and Apache Cassandra[12], make use of LSM-trees to offer an extremely effective storage backbone per column family. Wide-column stores' excellent scalability, extensibility, and support for Map-Reduce tasks make them suitable for analytical applications[13]. (For the concurrent processing of big datasets). Wide-column stores are difficult to employ for applications with changing schemas due to the predefined set of column family stores [14, 15].

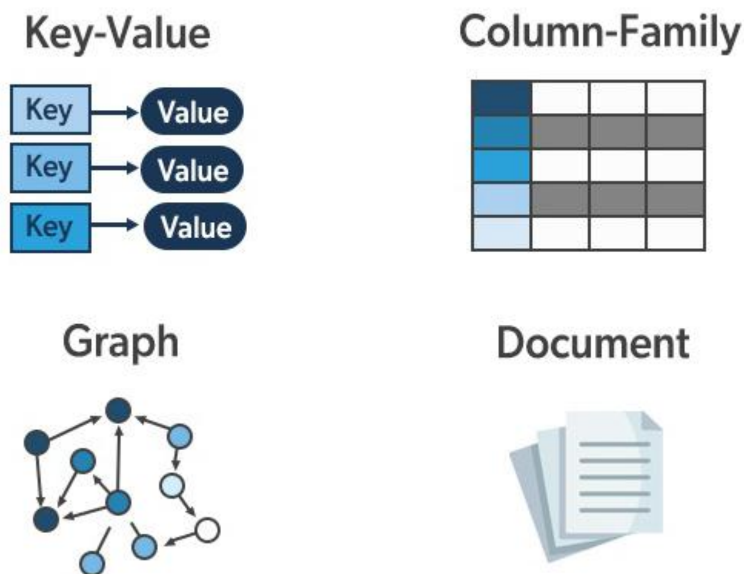


Figure 2.1 shows the ***NoSQL*** *Data-Models*

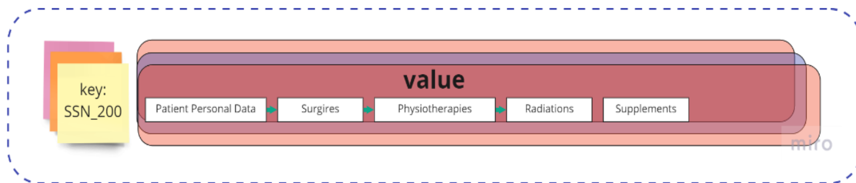
2.2.3 Document Data Model

They go beyond key-value storage, with the value component being nested documents. Key-value pairs with a flexible syntax are encoded using common semi-structured formats like XML or JSON. Each property in a document has a name and one or more values. A document is a collection of properties. The value component might be straightforward or complex, such a list of attributes or an embedded document. The extensible schema of a document may be kept up to date by adding or removing properties during runtime. As opposed to the opaque substance of values in Key-value stores, document stores comprehend the structure of documents and offer indexes and search operations based on their property names and values.

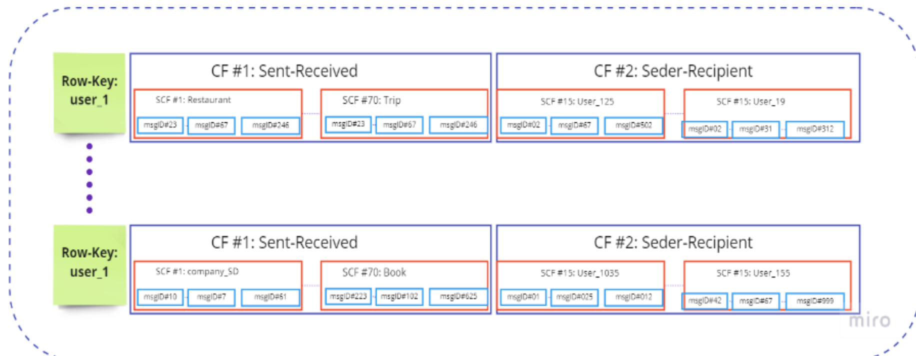
Programs like blogging platforms and content management systems (CMS) that may simply express their document storage works well with data in document format. For instance, a blog post with several (nested) features, such as tags, comments, pictures, and videos, may be simply represented in a document format. These stores are suitable for the modern Web 2.0 applications' high development efficiency and low maintenance expenses as a result of two important aspects. Many applications benefit from the adaptable data model of document storage since their data format is always changing. Consider a monitoring software that gathers, records, and examines log data from a variety of sources, each of which generates distinct data. It is straightforward to adapt to new log formats because of the document design's flexibility. Yet, because it calls for the creation of fresh tables for fresh forms or the incorporation by adding additional columns to the existing tables; in relational databases, such an extension would be costly. Web 2.0 applications make use of JSON-based data formats and tightly integrate Python, Ruby, and JavaScript. The impedance mismatch [16] between these programming languages

and document repositories has greatly decreased [17] due to the ease of transferring object-oriented approaches to documents. A collection, sometimes known as a bucket, is a special construct made available by a number of document stores, including Couchbase Server and MongoDB that consists of a group of documents that all represent the same kind of data. In that each row in these collections represents a document with a unique key but not necessarily the same structure as other rows, they resemble relational database tables. Instead, consider factors like as resources, replication and durability. When using collections, security can be managed for each set of documents. Figure 2.2(c) shows how Registration Students Record used the document data model to develop an interactive learning site that gives tailored search results. It combines Couchbase Server with ES for full-text search and content discovery. According to the Register Students Record vision, a textbook is separated into media pieces including articles, images, and videos.

(a) Using A key-value data model for Electronic Health Record (EHR)



(B) Using the wide-coulmn data model for searching about Facebook profile



(C) Using a document data model in Sutdent Registration Record

(D) Using a graph data model for Social Network of facebook.

an illustration, suppose Ali along with his friend Sarah visit the Eiffel Tower. Ali uses his mobile phone to record this visit by using "cheching in" features to the Eiffel Tower and tagging Sarah to let other friends know that she is also there. Osman Write a comment on this and Afraa likes it

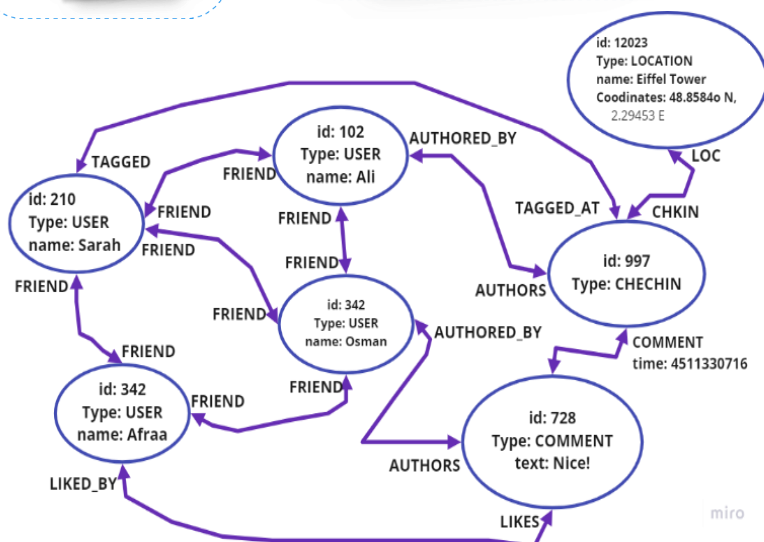


Figure 2.2 shows the case studies of ***NoSQL*** *Data Models*

1) *Content Metadata*, which holds the metadata of media assets with the content of text articles, is one category of data saved in JSON documents.

2) *User Profiles*, which keep track of each media object's user views and are used to modify ES search results based on user's settings or preference, and

3) *Content Stats*, which tracks viewer data for each media asset and uses that data to improve ES search results depending on the popularity of particular documents.

Document storage allows for data searching within documents without having to obtain the complete document and examine it at the application level. For instance, the following query demonstrates how to use N1QL10, a SQL-like language for Couchbase, to seek for documents in the collection *Content Metadata*. As seen in Figure 2.1. (c). after sorting the documents by property title with the value "Lesson One," the search returns the URL and the categories' property values.

Remember that native XML stores served as the inspiration for conventional JSON document storage. They use a variety of XML tools and standards for XML display, storage, keyword search, query processing, and optimization[18],. However many applications have selected JSON as an alternative to XML because of its resemblance to simplicity, tight compatibility with programming languages, and resemblance to compactness. Both XML and JSON storage in this regard offer a variety of applications and use-cases. While XML stores are often used for organizing and archiving a collection of XML files in content management systems such as those used in the health care, science, and digital libraries, JSON stores are employed by more interactive and dynamic Web applications.

2.2.4 Graph Data Model

In the abovementioned data structures, entity-related data is stored as binary digits, wide-column table rows, or documents. The emergence of graph stores, however, has been fueled by the expansion of graph-oriented datasets, such as the semantic web[19], web mining , and the interaction of proteins in biological systems[20, 21], which are efficient at storing such datasets and querying entity relationships. These databases are based on the graph theory, which states that a graph is composed of vertices that stand in for entities and edges that indicate relationships between those entities. Table 2.1 lists a number of graph structures that are frequently not mutually exclusive. For instance, directed, labeled, attributed, and multi-graphs are all combined in property graphs, which are often used in practice. Property graphs are popular because they may represent a variety of structures. For instance, when attributes are not utilized in property graphs, RDF or semantic graphs are formed. An RDF graph is a set of RDF assertions (or triples) that together represent a simple relationship between two entities. Facebook uses a property graph as an example of a social network in Figure 2.2(d), where users, real-world locations, relationships (like friendships between users), and actions (like liking, commenting on, and checking into a location) are all encoded using labeled vertices and labeled directed edges of a graph. Each edge has a triple of source, destination, and edge label vertex identifiers, and each vertex has a unique vertex identifier. For each edge label, a set of Key-value pairs representing other properties are also supplied, including the time property. All of the edges in Figure 2.2(d) are bidirectional (either symmetric, like FRIEND, or asymmetric, like AUTHORS/AUTHORED BY), with the exception of the edges labeled COMMENT. This is so that the COMMENT vertex and CHECKIN vertex

do not need to be traversed. In Table 2.2, the characteristics of the mentioned ***NoSQL*** data models are listed.

Table 2.1 Graph Model Structure

Graph Structure	Descriptions
Direct/Indirect Graphs	Any interaction in a network that serves no purpose is symmetric.
Labeled Graphs	Vertices and edges are given scalar values (labels or types) to identify their roles in various application domains or other relevant data.
Attributed Graphs	A changeable collection of attributes is given when KV pairs are connected to vertices and edges, representing their characteristics. Platforms for social networking that encourage human interaction are suited for it.
Multi Graphs	Self-loops and numerous edges between the same two vertices (even ones with the same labels) are likewise acceptable.
Hyper Graphs	These networks may represent N-ary relationships by using hyper-edges, which can connect any number of vertices.

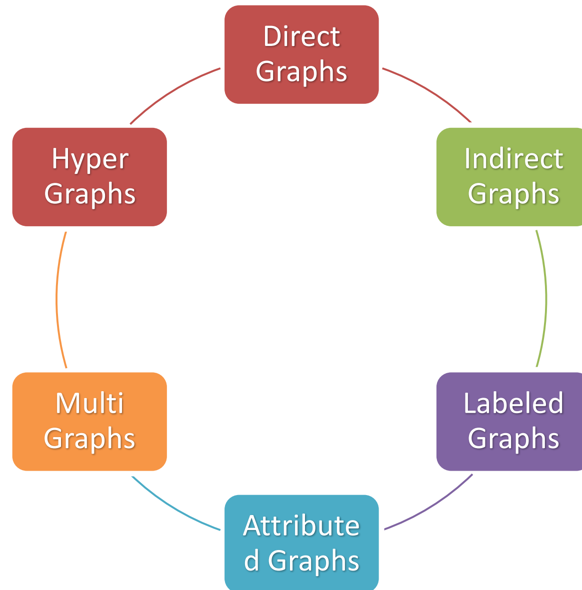


Figure 2.3 shows the types of graph model

2.3 Advantages of *NoSQL*

NoSQL is advantageous over other database methods due to its ability to accommodate changes for future upgrades and no need to depend on SQL functions or operations. ***NoSQL*** can make use of binary objects or data files in JSON/ XML formats, and soothes the workflow when the project development is in agile methodology.

NoSQL has five most salient benefits, which are as follows: schema-less, dynamic schema, structure of nested objects, increment methodologies, array features that may be indexable, and scaling out database. These benefits make ***NoSQL*** Databases a unique stance when compared with other types of databases.

The term "massive data" has become increasingly popular due to the expense of scaling up large data. New types of databases, such as ***NoSQL*** databases, have emerged to support scaling out and reduce administration and performance requirements.

NoSQL databases are typically built from the ground up to eliminate unnecessary managements and easier data models. They also allow for applications to keep virtually any structure, from less flexible elements to more rigorously defined ones. Additionally, ***NoSQL*** databases make it simple to add new columns because changing the schema of a ***NoSQL*** database does not require a laborious change process.

MongoDB is a flexible and easy to use database for programmers, but it has its own quirks. Cloud computing platforms have different licensing economics, so it is important to make apples-to-apples comparisons. Funding may be swayed by the startup's capacity to scale, and there are a lot of venture-funded ***NoSQL*** businesses. Groupthink is present.

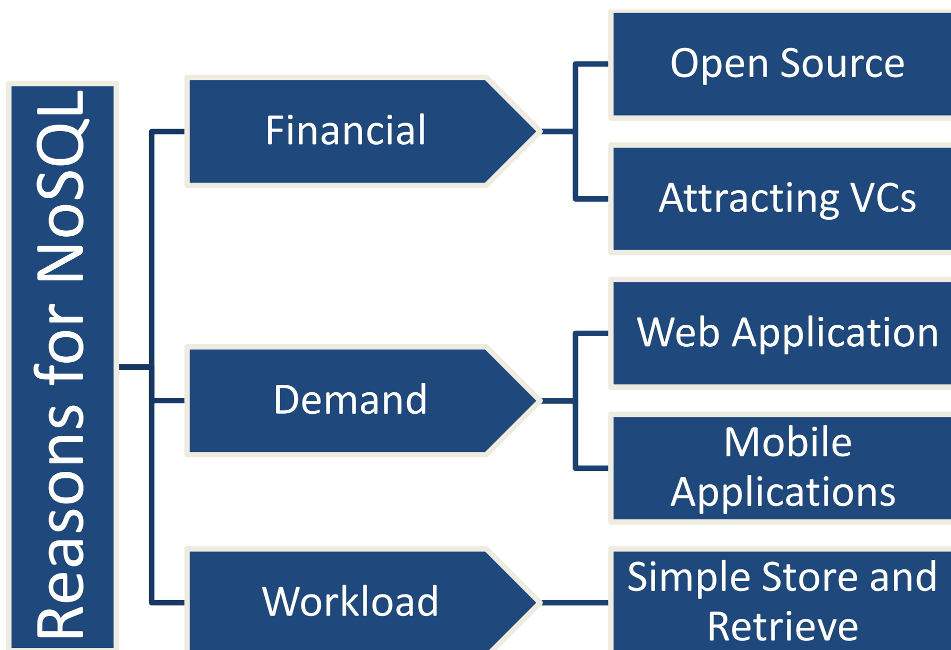


Figure 2.4 ***NoSQL*** *Reasons*

NoSQL databases are designed to be highly scalable and distributed systems, making them well-suited for concurrent algorithms. Concurrent algorithms allow multiple operations or queries to execute simultaneously on different parts of the data set. This can help improve performance by allowing more efficient use of resources and reducing latency. Examples of common concurrent algorithms include replication, sharding, and caching. Replication involves duplicating data across multiple nodes so that it can be accessed faster. Sharding divides large datasets into smaller chunks which can then be stored on separate nodes. Caching stores frequently used information close to where it will be needed, such as in memory or on disk storage. By utilizing these techniques, developers can ensure their applications have access to up-to-date data while minimizing resource contention. Additionally, they can also reduce network traffic since requests do not need to travel far between nodes. As a result, concurrent algorithms provide an effective way to optimize performance when working with ***NoSQL*** databases.

2.4 *NoSQL* Data Partitioning

Data partitioning is a technique used in ***NoSQL*** databases to divide data into independent partitions and distribute them among storage nodes. Horizontal partitioning separates the data into a number of discrete partitions or shards at the row-level, while vertical partitioning creates a number of separate partitions from the data that is often accessed together. Partitioning data offers certain benefits, such as increasing scalability, boosting system performance, and preventing single point of failure. Sharding can be key-oriented or solely based on data search.

A database is partitioned when it is divided up into smaller units called partitions. Partitions come in two varieties: vertical partitions and horizontal partitions. A database

table is divided vertically along the column characteristics, whereas a database is divided horizontally along the rows. Each server that oversees a particular portion of the database is referred to as a database shard when a single database is partitioned and shared (or scaled) among several servers. This procedure is referred to as sharding. Partitioning, which includes sharding, involves dividing the database into several segments (or sections). When a database is partitioned, a separate server may or may not be used to handle each portion of the divided database. But with sharding, each segment is overseen by an independent server

- **Sharding**: however, allows for the management of each portion by a different server. Hash partitioning, Range partitioning, and Robin-round partitioning are the three different forms of partitioning.
- **Hashing**: Implementing hashing partitioning involves applying a hash function to each data key. The node that houses the key would be identified by the hash function's result. Applications that often employ random scans should use hash partitioning. The hash function is applied to the data's key in order to locate any data item. The location of the data item would be revealed by the outcome.
- **Range partitioning**: Each node keeps a unique range of data keys in range partitioning. Applications that primarily need sequential scan can use range partitions well since most data items' keys are sequential.
- **Range partitioning**: Each node keeps a unique range of data keys in range partitioning. Range partitions function well with applications that primarily need sequential scan since the majority of data items with closely related keys are kept on the same node.

- **Round-Robin Partition**: In this method, the number of nodes determines how evenly the key items are spread (in a ring-like pattern). If there are three nodes, for instance, key items 1 through 3 will be shared over nodes 1 through 3, key items 4 through 6 will be spread across nodes 1 through 3, and so on.

2.4.1 Scaling

Vertical scaling and horizontal scaling are the two forms of scaling that are used to expand a database node's processing capacity. In vertical scaling, a machine's processor count, memory capacity, and disk space are expanded to facilitate the processing of additional data. On the other side, horizontal scaling refers to the addition of new hardware or an increase in the number of nodes used for data processing. The justification for horizontal scaling is a two-dimensional one. First of all, increasing the hardware will speed up work completion. There is a limit to how far a single node may scale vertically in vertical scaling. In addition, recent research has demonstrated benefits of parallel processing (horizontal scaling). Moreover, recent research has demonstrated that Moores law, which claims that "the number of transistors on a microprocessor chip will double around every two years," is quickly becoming unrealistic, supporting the need for parallel processing (horizontal scaling). Because of this, big data processing applications favor horizontal scaling over vertical scalability. Relational databases are made to scale vertically, whereas ***NoSQL*** databases are made to scale horizontally. As there is no restriction on the number of nodes that may be added to the database cluster, this gives ***NoSQL*** databases a bigger benefit.

2.4.2 Replication

The technique of keeping several copies of a database in various places in order to offer fault tolerance is known as replication. Higher levels of availability are the consequence, but a new set of difficulties are also presented. There are two forms of replication: eager (synchronous) and lazy (asynchronous) replications [49]. Replicas are updated during a transaction in eager replication, whereas they are updated afterwards in lazy replication. Maintaining consistency across the copies poses a number of challenges and may require making trade-offs. Eager replication affects performance, uses more bandwidth, and prolongs transaction delay. On the other side, lazy replication might result in some clones having outdated and stale data?

In systems where a high level of data consistency is required, outdated copies might not be permitted to respond to client queries. A database system's implementation of replicas has conflicting effects on the system as a whole. For instance, a large number of clones suggests that the system can offer more failure tolerance. A large number of copies also suggests that more work (in terms of bandwidth and delay) will be required to maintain the consistency of the replicas. Developers employ a variety of strategies and techniques to streamline their replication procedures. One of these methods is primary-secondary replication, wherein only a primary replica can handle writes while secondary copies can only process reads (or updates).

To ensure consistency among replicas, several quorum or consensus procedures are also utilized. The replication model used depends on the requirements of the individual application. All ***NoSQL*** databases employ replication in some capacity for fault tolerance. Consistent hashing and an eventual consistency model are used by Dynamo for replica

placement and maintenance, respectively. To synchronously handle writes across replica, Google Megastore makes advantage of Paxos.

Algorithms for failure detection are present in most clusters. The gossip-based protocol used by Dynamo. Between the master server and slave servers, BigTable employs heartbeat messages. A node is believed to have failed if it doesn't respond within a certain timeout period. Each of these systems employs different methods for recovering a failing node once its faults have been detected.

2.4.3 Load Balancing

In order to prevent any one node from becoming overwhelmed, load balancing is a strategy used to manage, distribute, and re-distribute data between nodes. High throughput, effective resource use, minimal latency, and the avoidance of hotspots across the cluster are the goals of load balancing and distribution. In a cluster system, load balancing also seeks to enhance fault tolerance and maximize the replica distribution procedure. For instance, a load balancing approach called shard aware or rack awareness makes sure that replicas are spread so that network outages on a single rack do not influence availability. Using two copies on two separate nodes in a local rack and the third replica on a different node in a different rack is a well-known rack-aware strategy used in HDFS. In this manner, a network partition to a rack won't be impacted.

2.4.4 Garbage collection

To ensure availability, even at the price of consistency, the majority of cloud databases frequently maintain several copies of every data item. To efficiently manage computer resources (storage/memory) and stop the database from storing useless or superfluous data, an efficient garbage collection procedure is required. In order to ensure that only

logs that would not be needed are destroyed, log files must be trash collected carefully. The system's efficiency shouldn't be hampered by the rubbish collecting procedure. While the master is in a quiescent state, garbage collection is done in batches. While managing data, the aforementioned methods are applied in varying degrees. The whole of these methods combined, used by a database management.

2.4.5 Recovery and Identification of Failures

Failure is commonplace in the cloud computing context, as was previously described. This is due to the fact that hundreds to thousands of machines—most of which are common commodity machines—are utilized to process data in parallel. To ensure availability and consistency, a reliable method must be in place to identify failing equipment and bring them up to date.

2.4.6 Transaction Management

The execution of transactions must be serializable in order to maintain data consistency. A transaction execution that can be serialized is one whose output would have the same outcome as if the transactions were run sequentially. A scheduler is employed in relational databases to guarantee that transaction executions are serializable. Prior to beginning, a transaction obtains locks on all the data objects involved and maintains the locks until all activities within the transaction have been completed. A transaction releases all locks after performing the operations. A transaction obtains locks for all of the data objects involved in the transaction in the first phase. The transaction is not authorized to request more locks once locks have been released.

The transactional management approach is one of the primary methods used in database systems to preserve the consistency of shared data throughout the concurrent execution

of many requests originating from various users. Several read and write operations are combined in database systems into (atomic) transactions that adhere to ACID principles (Atomicity, Consistency, Isolation, and Durability). The execution of transactions must be serializable in order to maintain data consistency. A transaction execution that can be serialized is one whose output would have the same outcome as if the transactions were run sequentially. A scheduler is employed in relational databases to guarantee that transaction executions are serializable.

Locking is a method that is typically combined with a scheduler. Prior to beginning, a transaction obtains locks on all the data objects involved and maintains the locks until all activities within the transaction have been completed. No other transaction may alter the data while it is locked during this time. The isolation of transactions from one another would be ensured by doing this. A transaction releases all locks after performing the operations. Two phase locking is the name of this procedure (2PL). A transaction obtains locks for all of the data objects involved in the transaction in the first phase. The second phase involves the release of all obtained locks.

The transaction is not authorized to request more locks once locks have been released. Concurrency control is the process used to make sure that transactions always leave the database in a consistent state. Nevertheless, in a distributed database system, the method for concurrency management becomes more complicated. Each participating database's schedulers are in charge of controlling the portion of data that it stores.

2.5 CAP Vs. BASE

Rothnie et al. were the first to observe this trade-off. They increasing commercial appeal of the Web, as well as the growing demand for regional data replication and high

operational availability[22], group Fox and Brewer [23, 24] they recover this trade-off as a CAP rule. Concurring to this run the show, a dispersed information capacity can as it were fulfill two out of the three alluring properties of consistency, accessibility, and segment resilience at once.

The CAP theorem was later codified and proven by Gilbert and Lynch[24, 25], and defined in this context as follows:

1. **Consistency**: It is believed that linearizability is shown by consistency, a qualitative quality. This hypothesis assumes that the CAP consistency is not a metric that is continuously tracked in response to the system's operational condition. The linearizability of the system's used algorithms is therefore statically specified with respect to those algorithms, as evidenced by the consistency.
2. **Availability**: The availability property implies that every request a client makes will ultimately (within a predetermined amount of time) result in a successful (non-error) response. According to this theory, a system's "availability" or "unavailability" is statically stated in relation to the algorithms it employs. Yet, there are a few questions around this concept. Prior to anything else, it's possible that certain highly available systems with high uptime rates aren't CAP-available. A distributed data store using quorum-based synchronous replication, for instance, is not CAP-available because read/write operations on the minority side of the partition might not succeed when the network divides. There is no upper limit on response time, second. For instance, an operation is considered to be CAP-available if it is successfully finished after one week. But according to the core concept of accessibility, such a method is not accessible. In other words, this formulation does not account for the realistic aspect of (latency) (reaction time).

3. **Partition tolerance**: This is regarded as a qualitative attribute and indicates that the system keeps up with its CAP-availability or CAP-consistency promise even in the face of a network partition (of the algorithms employed by the system). Given that distributed systems can fail in ways other than network splits, this notion is nebulous and imprecise. To put it another way, additional problems exist, such as message loss and node failures.

The distributed databases are categorized into the following subcategories by giving up any one of the aforementioned CAP properties.

1. **Database systems with Consistency and Availability**: The algorithms that CA systems employ make no putative divisions of networks. It is consequently nearly difficult to create this combination in distributed systems since network splits would always occur. The major CAP trade-off, thus, is between availability and consistency. More consistency than availability is compromised in scattered data storage, hence this trade-off has evolved into a defense for accepting uneven consistencies.
2. **Database systems that provide consistency and partition tolerance**: This Distributed data storage that upholds CAP-consistency enables combination. A read/write request might not be fulfilled, nevertheless, in order to lessen the chance of consistency problems in the event of a network split. CP makes sense for systems designed to operate in a reliable network, such as a single data center, because network partitions are uncommon.
3. **Availability and tolerance for partitions Database systems**: Distributed data storage with lax consistency enforcement enables this combination. Nevertheless, conflicting writes are permitted to be finished, which might cause clones to

diverge and calls for the creation of a dispute resolution mechanism. These systems are generally employed by wide-area network applications, such web caching, whose users must have a high level of availability and quick response times.

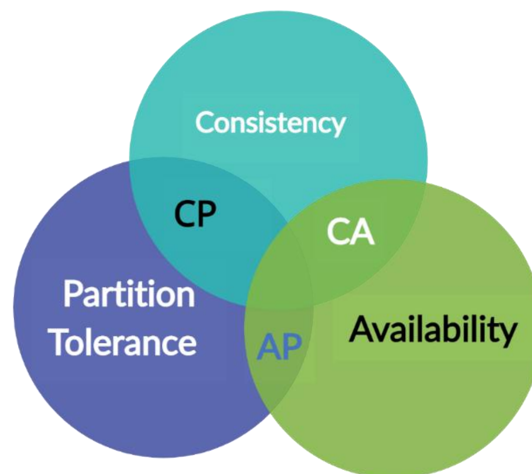


Figure 2.5 shows the CAP Characteristics' of distributed databases

BASE

Although it integrates with practice, BASE theory is a byproduct of CAP theorem and is totally distinct from ACID model. Fundamentally Available, Soft-state, and Eventual consistency is referred to as BASE. Partition failure might be provided, but basically availability is simple to understand. Soft state indicates that the system's state could become nonsynchronous over time. Finally, there should be uniformity in the data. One of the consistency models utilized in the field of parallel programming is eventual consistency. That means that all updates may be anticipated to ultimately propagate through the system and all the replicas will be consistent if there is a sufficiently enough period of time during which no modifications are delivered.

As a result of the aforementioned reasoning, the non-relational data storage trend known as ***NoSQL*** stores has developed with the intention of solving the high availability and scalability requirements of big data applications. The term "Not Just SQL" (or ***NoSQL***) references to several of these systems that provide SQL-like queries. According to Cattell, R [26, 27] contemporary database systems are referred to as "data stores," where more adaptive data models are used and DBMS functionalities may not always be fully utilized.

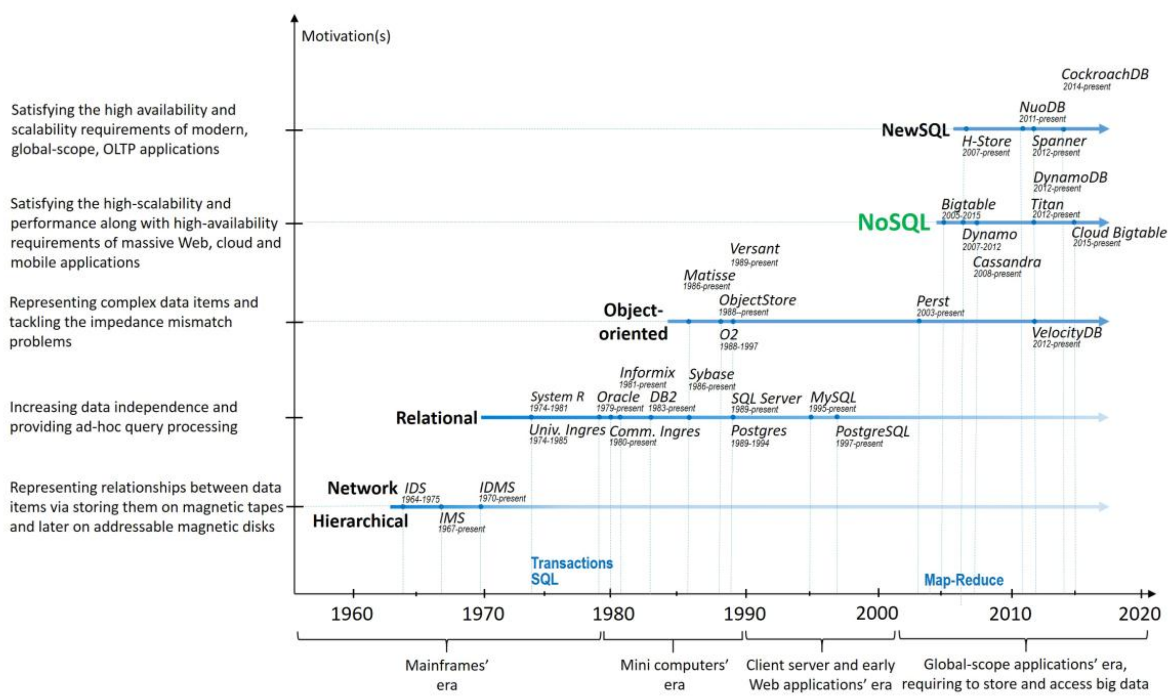


Figure 2.6 Technology timeline for primary databases and accompanying systems[28]

Large amounts of semi-structured and unstructured data are stored and managed using a form of database management system called ***NoSQL***. ***NoSQL*** databases do not require a set schema or predetermined data model, in contrast to conventional relational databases.

Key-value, document-oriented, column-family, and graph databases are just a few of the many data kinds and structures that ***NoSQL*** databases are made to manage. In large data and real-time online applications, where scalability, flexibility, and high availability are crucial, these databases are frequently utilized.

NoSQL databases have the potential to expand horizontally, meaning that more servers may be added to manage growing amounts of data and traffic. This is one of its key advantages. They are therefore highly suited for applications like e-commerce, social networking, and mobile ones that demand high performance and availability.

Another key feature of ***NoSQL*** databases is their ability to handle unstructured data, such as text, images, and video, which can be difficult to store and manage in traditional relational databases. ***NoSQL*** databases can also handle semi-structured data, such as JSON and XML, which are commonly used in web applications.

In addition to their scalability and flexibility, ***NoSQL*** databases also offer a number of other benefits, such as faster performance, simpler data modeling, and lower cost. However, they also have some drawbacks, such as a lack of standardization and limited support for complex queries.

Overall, ***NoSQL*** databases are a powerful and versatile tool for managing large volumes of data in a variety of applications. With their ability to handle diverse data types and structures, and their scalability and flexibility, they are well-suited for the demands of modern big data and web applications.

2.6 The Data Consistency in *NoSQL* Databases

NoSQL databases are typically designed to provide high scalability, flexibility, and performance for handling large volumes of unstructured or semi-structured data. However, ensuring transaction consistency in a ***NoSQL*** database can be challenging due to their distributed and decentralized nature. Below, I'll describe some common approaches and techniques used to achieve transaction consistency in ***NoSQL*** databases.

An operation sequence that usually complies with the ACID properties is referred to as a *transaction*. If a transaction is successful, it is said to commit; if not, it is called to abort[52]. A single valid state for all database instances can be characterized as *consistency* in database management systems ***DBMS***. A *database management system consistency* can be defined as a single acceptable state for all database instances as long as the data remain the same across all redundant database servers.[53, 54]. Because a DBMS must guarantee that the returned data is the most recent for readings and must confirm that the write operation has been successfully performed on each requested server, this has an influence on performance. Since there are more database replicas present in distributed systems, availability is also impacted by consistency policy in a manner similar to how accessibility is. In order to boost efficiency and availability, ***NoSQL*** databases use eventual consistency, which permits temporary inconsistency (i.e., not all redundant servers will immediately have the most recent data) and permits a database replica to return its available data (which may not be the newest). There will finally be consistency across all redundant servers[55]. ***NoSQL*** DBMSs permit the adoption of different consistency levels (i.e., the bare minimum of redundant servers holding the most recent data), which are adjusted in accordance with an application's needs[56]. This contributes to closing the inconsistency. Fewer servers need to be

upgraded because of fault tolerance and increased availability. Another ***NoSQL*** trait is strong consistency, which always returns the most recent data.

According to the most recent studies, consistency models can be categorized into a variety of categories, including strong consistency, weak consistency, eventual consistency, causal consistency, read-your-writes consistency, session consistency, monotonic reads consistency, and monotonic writes consistency.

1) **Weak-Consistency Model:**

   This model, as the name indicates, reduces consistency. It specifies that a read operation does not guarantee the return of the most recently stored value. It also does not ensure the sequence of events[57]. The time interval between a write operation and the point at which each read operation provides the updated data is referred to as the inconsistency window[58]. Because there is no need to include more than one replica or node in a client request, this paradigm results in a highly scalable system.

2) **Eventual Consistency Model:**

Eventual consistency is a common approach in ***NoSQL*** databases. It allows for data to be inconsistent temporarily but guarantees that, given enough time and no further updates, all replicas will converge to a consistent state.

This approach is often used in scenarios where immediate consistency is not a strict requirement, such as social media feeds or recommendation systems.

A consistency model that ensures if there is no additional updates on a given item, all the reads to that item will eventually return the same value[58]. Replicas frequently arrive with the same data state. Read operations might not always

return the most recent version while this procedure is in progress. The connection lags between replicas and their sources, system load, and the number of replicates involved will affect the inconsistency interval.[57]. This method is half-way between a strong-consistency model and a weak-consistency model. Many ***NoSQL*** databases provide Eventual Consistency as a feature. The world's most popular companies that use Cassandra can provide availability and network partitioning to such a degree that it does not hinder functionality. Facebook, the company that originally developed Cassandra, is one of them.

3) **Strong Consistency Model:**

The identical value will be returned by any read from any replica thanks to a robust consistency model. All clients will utilize the identical data entry and data, and each transaction must appear to be committed instantly. The write action must commit before a read operation may access the updated version of an instance. Every storage system instance accepts a particular global sequence of events. [58, 59].

4) **Casual Consistency Model:**

Any operations that recognize the update on an element are required to take the modified value into account. The eventual consistency model will be used in the event that another process does not acknowledge the write operation [18]. Although less dependable than sequential consistency, causal consistency is more dependable than eventual consistency. When the Eventual Consistency model is reinforced to be Causal Consistency, the system's availability and network partitioning properties are decreased.[57].

Causal consistency aims to maintain causal relationships between operations. It ensures that if one operation causally depends on another, the former will appear to have occurred after the latter.

This is valuable in scenarios where maintaining causality is critical, like distributed systems with complex dependencies.

5) **Read-Your-Writes Consistency Model:**

With the help of the read-your-writes consistency model, it is made sure that a replica is at least current enough to include changes made by a single transaction. Transactions are applied sequentially, therefore by guaranteeing that a replica has a particular commit applied to it, we can make sure that all transaction commits that took place prior to the given transaction have already been committed to the replica. If a process updates an object, that process will always take into account the modified value. Other processes will eventually read the modified value. Therefore, read-your-writes consistency is achieved when the system guarantees that every attempt to read a record that has been modified will return the updated value.

Some ***NoSQL*** databases, like Google Spanner, provide strong consistency guarantees. Strong consistency ensures that all replicas of the data will return the same value for a read operation, even in a distributed environment.

Achieving strong consistency often requires coordination mechanisms like two-phase commits, which can impact performance and availability.

6) **Session Consistency Model:**

A process will follow a read-your-writes consistency model for the length of a session if it makes a request to the storage system while it is operating within that session. All reads are current with the session's writes using session consistency,

although writes from other sessions may need to wait. Although everything arrives in the correct order from prior sessions, the data is not always guaranteed to be up to date. This offers excellent consistency at half the cost of good performance and availability.

7) **Monotonic Read Consistency Model:**

Every time a process reads a value, it returns that value or one that is more recent[54]. It implies that the same item is read by the same process consistently and in the same order. However, this does not guarantee that read operations between processes on the same object will be ordered monotonically. Because of this, monotonic readings ensure that a process that reads r1, r2, and r2 cannot experience a state that is earlier than the writing represented in r1; reads, by nature, cannot travel backward. Monotonic readings do not apply to operations carried out by different processes; they only apply to those carried out by the same process. There are full monotonic readings available: Even during a network split, all nodes can advance[60].

8) **Monotonic Write Consistency Model:**

Before any more write operations by the same process on the same object, a process-initiated write action on that particular object must be completed[58]. In other words, the same process writes to the same object consistently in the same order. However, this does not guarantee that write operations between processes on the same object will be ordered monotonically. The effect of this is that monotonic writes guarantee that if a process writes w1, then w2, then all processes will observe w1 before w2. Monotonic writes do not apply to operations carried out by different processes; they only apply to those carried out by the same

process. All monotonic writes are available: Even during a network split, all nodes can advance[61].

9) **Time-Line Consistency Model:**

Yahoo created this consistency model especially for YAHOO PNUTS in order to solve the inefficiencies of serializable transactions of the big data and its relation with their geo-replication. Furthermore, it seeks to reduce the shortcomings of eventual consistency[62]. ***NoSQL*** databases are support eventual consistency instead of strong consistency. They do not support database transactions which ensure strong data consistency[63].

10) **Quorum-Based Systems:**

Many ***NoSQL*** databases use quorum-based techniques. In these systems, a write or read operation is considered successful only if it meets a quorum (majority) of replicas. This ensures that a certain level of consistency is maintained.

For example, in a replica set with five nodes, a write operation may require a quorum of three nodes to acknowledge the write as successful.

Each type of ***NoSQL*** models support many level of Consistency for example the eventual consistency supported may levels of consistency For the confirmation of an activity at consistency level **ONE**, just one node or server is required (such as a write or read). For level 2 operations, **TWO** nodes are needed, and while reading, the most current data from both servers is taken into consideration. The **QUORUM** policy[64], which requires that the least integer bigger than 50% of the database nodes be used to determine consistency, is compatible with a level like this. Like the **ALL** policy, Level Three asks confirmation from each

node.[65]. The latest information is constantly accessible thanks to reading (high consistency)[66].

11) **Conflict Resolution:**

In multi-master ***NoSQL*** databases, conflicts can arise when concurrent writes occur to the same data. Conflict resolution mechanisms, like "last write wins" or custom resolution logic, can be used to resolve these conflicts and maintain data consistency.

12) **Vector Clocks and Versioning:**

NoSQL databases often use vector clocks or versioning to track changes to data over time. This allows the database to determine the order of operations and resolve conflicts when they arise.

13) **Tuning Consistency Levels:**

Some ***NoSQL*** databases allow you to configure the level of consistency on a per-operation basis. You can choose between options like "strong," "eventual," or "session" consistency to balance performance and data integrity based on your application's requirements.

14) **Distributed Transactions:**

Some ***NoSQL*** databases support distributed transactions. In this model, transactions can span multiple documents or entities, and the database ensures that either all changes in the transaction are committed or none are.

Distributed transactions often require careful handling of distributed locks and can introduce latency and complexity.

Table 2.2 *NoSQL* Consistency Models

Consistency Model	Grantees	
Weak Consistency Model	A read operation will not really support serialization and doesn't guarantee that it will provide the value that was most recently saved in memory.	
Session Consistency Model	Consistency with read-your-writes is only guaranteed during a session.	
Read-Your-Writes Consistency Model	An operation always receives the most recent update on read operations.	
Monotonic Reads Model	Every time return the same value as the last reading, or one that is more recent.	
Monotonic Writes Consistency Model	Prior to performing any more writes, a write operation must always complete.	
Casual Consistency	Order of actions overall with a causal connection	
Strong Consistency	Serializability	A set of operations is composed of concurrent computations of a group of serialization units.
	Linearizability	Every operation is immediately seen in the overall, sequential order of events, or it is handled as a single operation.
Eventual Consistency	Eventually, the state of the updates will be consistent across all replica nodes.	
Time-line Consistency	The actions are performed on the same record by all replica nodes in the same "correct proportion".	

Table 2.3 Consistency Models in ***NoSQL*** Databases

***NoSQL* Database**	**Data Model**	**Consistency Model**	**Applications/Services**	**API**
Dynamo	Key-Value	Eventual Consistency	E-Commerce Platforms like Amazon Stores (AWS Amazon Web Services)	Multiple Consistency Level
Cassandra	Column-Family	Eventual Consistency	Facebook, Netfelx, inbox search, eBay, Sound Cloud, Rack Space Cloud	Multiple Consistency Level (ONE, TWO, ALL, QURAM)
Raven DB	Document	Eventual Consistency	Toyota	Multiple Consistency Level
MongoDB	Document	Eventual Consistency	SAP AG Software Entreprise, MTV, Vodafone, AMAR BANK	CRUD API
Raik	Key-Value	Eventual Consistency	Yammer Social Network, Github	Multiple Consistency Level
Yahoo PNUTS!	Multi-Model	Timeline Consistency	Yahoo Mail	Multiple Consistency Level
Apache HBase	Column Family	Strong Consistency	Facebook messenger, using Hadoop for large set of application	JSON API
Microsoft Azure	BOLB Tables	Strong Consistency	Office 365, OUTLOOK, Bing	RESTfull API
Redis	Key-Value	Strong Consistency	Flicker, Instagram	JSON API
Google Spanner	MultiModel	Strong Consistency	Google F1	SQL-Like

Many applications demand either a rigorously strong type of consistency or just static eventual consistency. However, consistency requirements are not evident for another type of applications since they are dependent on data access behavior dynamical, client

demands, and the results of reading inconsistent data such as ecommerce platforms because these kinds of applications, the fast accessibility and availability are critical. Strong consistency techniques may therefore be unaffordable. Although they are preferred for some applications, great levels of uniformity are not always required. In situations like these, undesirable results are caused by either immobile eventual or strong sorts of consistency. When storage systems are dispersed geographically, strong consistency guarantees become unaffordable due to high network latencies. As a result, applications requiring high availability and performance are best served by weaker consistency semantics, such as eventual consistency.

Conclusion

Remember that achieving strong consistency in a ***NoSQL*** databases often comes at the cost of increased latency and reduced availability, as it may require coordination between nodes. Therefore, the choice of consistency model should align with your application's specific needs and tolerance for trade-offs between consistency, availability, and partition tolerance (the CAP theorem).

2.7 NoSQL Transactions

NoSQL databases have gained immense popularity in recent years due to their ability to handle massive volumes of unstructured and semi-structured data. However, one of the persistent challenges in the world of ***NoSQL*** databases is maintaining data consistency when performing transactions. In this discussion, we will delve into the complexities of ***NoSQL*** transactions consistency, the challenges it presents, and some solutions to address these issues.

NoSQL databases are designed to handle large-scale data storage and retrieval requirements that cannot be efficiently handled by traditional relational databases. One of the key differences between traditional relational databases and ***NoSQL*** databases is the transaction model used for data consistency. This chapter discuss the models of ***NoSQL*** transactions

A transaction is described as the execution of a set of instructions that includes many actions, such as "read" or "write and update," to access and modify data in a database. To make the most use of the computational resources, these operations in transactions in typical relational databases are carried out by concurrent levels in an interleaved manner[80]. Hence, transactions must adhere to the ACID guidelines in order to verify the accuracy and integrity of a database (Atomicity, Consistency, Isolation and Durability). Atomicity refers to the requirement that every operation in a transaction be completed in its whole, failing which none of the actions must be performed. Consistency requires that the database remain in a usable condition following the conclusion of a transaction. In order to boost performance, durability and isolation both ensure that transactions do not conflict with one another and that changes made following a transaction are kept permanently in the database[81].

NoSQL databases are, however, unsuitable for usage in particular application areas that need robust data consistency, such as commercial or banking applications, because of the adverse impacts of this paradigm and design change on data consistency. This is because a traditional trait of relational databases, support for transactions, is not provided by ***NoSQL*** systems. In order to manage the enormous amounts of data, often known as "Big Data," efficiently, many cloud computing and ***NoSQL*** database companies have created various models and strategies. Most of the methods now in use concentrate on increasing productivity and data accessibility, but they pay little attention to guaranteeing data consistency.

There are two main transaction models used in databases:

1- ACID Transactions
2- BASE Transactions

2.7.1 Understanding NoSQL Transactions

NoSQL databases come in various flavors, including document stores, key-value stores, column-family stores, and graph databases. Each type offers different trade-offs in terms of scalability, flexibility, and performance. However, one common challenge across all ***NoSQL*** databases is ensuring data consistency when multiple operations are involved in a transaction.

2.7.2 Challenges in NoSQL Transactions Consistency

1. CAP Theorem: The CAP theorem, coined by Eric Brewer, states that it is impossible for a distributed system to simultaneously provide Consistency, Availability, and Partition Tolerance. ***NoSQL*** databases often sacrifice strong consistency for improved availability and partition tolerance. This means that some ***NoSQL*** databases may not provide strict ACID (Atomicity, Consistency, Isolation, Durability) guarantees.

2. Eventual Consistency: Many ***NoSQL*** databases embrace eventual consistency, which allows for temporary inconsistencies between replicas or nodes in a distributed system. While this approach can lead to better availability, it can make transactional data operations more challenging to manage.

3. Concurrency Control: Ensuring proper concurrency control in ***NoSQL*** databases can be complex. Optimistic and pessimistic locking strategies may not always work as expected due to the distributed nature of ***NoSQL*** databases, leading to issues like race conditions and deadlocks.

2.7.3 Solutions to NoSQL Transactions Consistency

1. ACID Compliance: Some ***NoSQL*** databases, particularly NewSQL databases, aim to provide strong ACID guarantees. These databases combine the scalability of ***NoSQL*** with the consistency of traditional relational databases. Examples include Google Spanner and CockroachDB.

2. Use of Consistency Models: ***NoSQL*** databases often implement various consistency models such as "read-your-writes," "session consistency," and "monotonic reads" to offer developers more control over data consistency. Developers can choose the level of consistency that best suits their application's requirements.

3. Conflict Resolution: Implementing robust conflict resolution strategies is essential in ***NoSQL*** databases. Techniques like version vectors, vector clocks, and last-write-wins can help resolve conflicts that arise due to eventual consistency.

4. Transactions as Code: Some ***NoSQL*** databases provide APIs that allow developers to define complex transactions as code. This approach enables developers to encapsulate multiple operations within a transaction, ensuring they either all succeed or all fail.

5. Hybrid Approaches: Hybrid databases that combine ***NoSQL*** and SQL capabilities are gaining popularity. These databases provide the flexibility of ***NoSQL*** for unstructured data while allowing developers to use SQL for structured and transactional data.

2.7.4 ACID Characteristics

As a method to guarantee the accuracy and consistency of a database that makes each transaction a combination of actions that functions as a separate unit, provides consistent results, works independently of other processes, and permanently stores any modifications it makes.

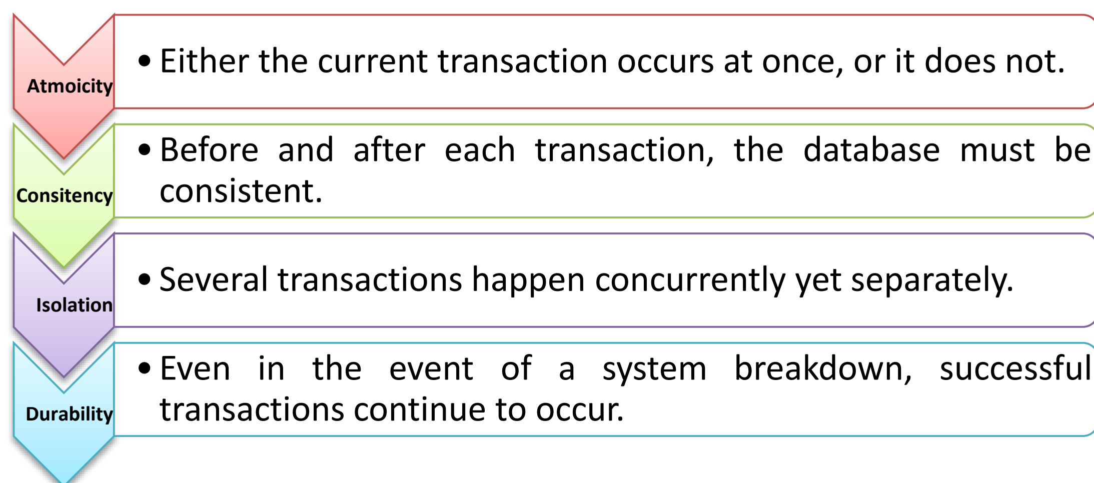


Figure 2.7 ACID CHARACTERISTCS

- **Atomicity** means that the full transaction either occurs all at once or not at all. There is no middle ground, hence partial transactions do not take place. Each transaction is treated as a single entity and is either carried out entirely or not at all. The following operations are involved.

    1. **Abort**: If a transaction fails, any database modifications are invisible. Commit: A transaction becomes visible once it has committed any modifications.

2. **Begin**: This establishes where a transaction's initial set of activities will take place. Operations can be either read or written.
3. **Commit**: At this point, all actions are completed and committed atomically, which requires that all operations between "begin" and "commit" be successful. Any operation in this scope that fails would result in a rollback operation.
4. **Rollback**: When one of the activities in a transaction fails, the rollback operation is initiated. This implies that in order to return the database to its starting state, all activities that have been carried out must be undone.

The "All or nothing rule" is another name for atomicity.

- **Consistency**: To ensure that the database is consistent both before and after the transaction, integrity requirements must be upheld. It speaks to a database's accuracy.
- **Isolation**: This characteristic makes guarantee that numerous transactions can take place simultaneously without causing the database state to become inconsistent. Transactions take place without interruption and independently. Modifications made in one transaction are not visible to changes made in any other transaction until the change in question has been committed or written to memory, whichever comes first. This property guarantees that the state created by simultaneously running transactions will be the same as the state created by serially running them in some sequence.
- **Durability**: This feature makes sure that when the transaction has finished, the database updates and adjustments are saved in and written to disk and that they endure even if a system failure takes place. These modifications are now kept

in non-volatile memory and are permanent. Hence, the transaction's consequences are never lost. When transactions are completed, the consistency property makes sure that the database is still consistent. Isolation makes sure that even if two transactions are carried out simultaneously, the result would be the same as if they were carried out serially, one after the other. Transactions must be serializable in order for them to be segregated from one another. Data loss might be avoided and failures could be recovered from using durability

The DBMS uses certain methods, which are covered later in this chapter, to preserve these ACID features. Before continuing, the idea of serializable transaction execution and a few abnormalities brought about by non-serializable transaction execution are described below.

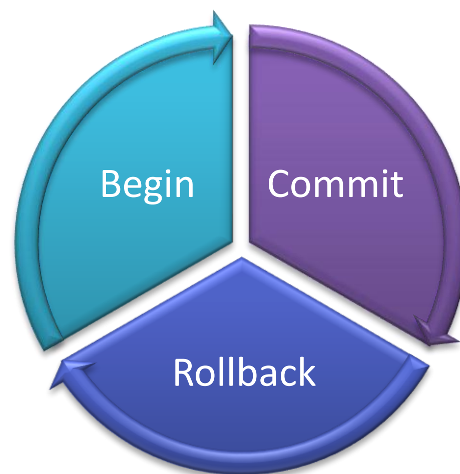


Figure2.8 the Transactional Life Cycle

ACID stands for Atomicity, Consistency, Isolation, and Durability. ACID transactions are characterized by their strict consistency and reliability. In an ACID transaction, a group of operations are treated as a single unit of work, and either all the operations in the group are executed or none of them are executed.

ACID transactions are commonly used in traditional relational databases, where data consistency is of utmost importance. However, in ***NoSQL*** databases, ACID transactions can be too restrictive and can negatively impact performance.

2.7.5 BASE Transactions

BASE stands for Basically Available, Soft state, eventually consistent. Unlike ACID transactions, BASE transactions prioritize availability and scalability over strict consistency. In a BASE transaction, it is possible for some operations to fail or for data to be temporarily inconsistent. However, the system eventually converges to a consistent state.

BASE transactions are commonly used in ***NoSQL*** databases, where data consistency is not as critical as in traditional relational databases. BASE transactions allow for greater scalability and availability, making them ideal for applications that require high performance and availability.

In conclusion, choosing the right transaction model is critical when designing a ***NoSQL*** database. ACID transactions are suitable for applications that require strict consistency and reliability, while BASE transactions are ideal for applications that prioritize availability and scalability.

2.7.6 ACID, CAP vs. BASE in ***NoSQL*** Databases

ACID vs. BASE

When it comes to transaction models in ***NoSQL*** databases, there are two main options: ACID and BASE.

ACID transactions are characterized by their strict consistency and reliability, which makes them ideal for traditional relational databases. However, in ***NoSQL*** databases, ACID transactions can be too restrictive and can negatively impact performance.

On the other hand, BASE transactions prioritize availability and scalability over strict consistency. This makes them ideal for ***NoSQL*** databases, where data consistency is not as critical as in traditional relational databases. BASE transactions allow for greater scalability and availability, making them ideal for applications that require high performance and availability.

CAP vs. BASE

CAP and BASE are two contrasting philosophies for building distributed systems. They represent different trade-offs between consistency, availability, and partition tolerance. CAP theorem guarantees two out of three properties, while BASE assumes that consistency can be traded off for availability and partition tolerance.

Here are some key differences between CAP and BASE:

1. Consistency: In CAP, consistency is prioritized over availability, which means that if there is an update in one replica, all other replicas must be updated with the same value before responding to the user. In BASE, consistency is relaxed, and it is accepted that replicas may be out of synchronization for a period of time while they are being updated asynchronously.

2. Availability: In CAP, availability is considered important, but it may be compromised to maintain consistency, which means that the system may not be able to respond to user requests when there are network partitions. In BASE, availability is a fundamental requirement and may involve relaxing data consistency guarantees to achieve it.

3. Partition Tolerance: In CAP, partition tolerance is a requirement that the system has to be robust and tolerate network partition and failures. In BASE, partition tolerance is also a requirement but it's more relaxed, and the system may cache or queue updates that can't be propagated to other nodes in case of network partition.

In summary, CAP emphasizes consistency and favors strong data integrity guarantees and real-time consistency, while BASE focuses on availability, scalability, and high performance, with eventual consistency as a compromise. The choice of whether to follow CAP or BASE depends on the specific requirements of the application, including the value of data consistency, business requirements, scalability, and performance.

2.7.7 *NoSQL* Transactional Databases

NoSQL databases are designed to handle large-scale data storage and retrieval requirements that cannot be efficiently handled by traditional relational databases. These databases don't rely on the traditional SQL-based relational model, which makes them more flexible and scalable.

2.7.8 Transaction as a services for *NoSQL* databases

NoSQL databases use different design principles than traditional relational databases in order to accomplish the goals of high availability, scalability, and efficiency.

The ***NoSQL*** Query Language's (NQL) operations have been condensed to Get/Put operations. As ***NoSQL*** databases promote availability over consistency, ACID transactions are not guaranteed

Several approaches have been set into implement transactions in ***NoSQL*** databases as a result of the significance of transactional services being accepted. Three potential

implementation levels, including the ***data store, middleware, and client side***, have been proposed for transactional services.

For the purpose of supporting transactions at the data store level, systems like Spanner[82], COPS, Granola[83], and Warp[84] have been created. Yet, this can jeopardize availability and scalability. Google Megastore[85] G-Store[86], Deuteronomy[87], CloudTPS[88, 89], pH[90], CumuloNimbo[91], and [92] are a few middleware strategies. By the use of middleware, which serves as the interface between clients and databases, these strategies create transactional services. Concurrency control and ACID aspects are therefore handled by middleware. The development of APIs that communicate and receive metadata from the client's apps is a component of the client layer strategy. Such examples are the system in[93], ReTSO[94], and Percolator[95].

The aforementioned approaches offer various degrees of consistency in ***NoSQL*** databases. They include sequential consistency and strong consistency or linearizability (global real-time ordering)[96, 97].

2.7.9 Concurrency Control Techniques in *NoSQL*

The core to a database system's performance is parallel of the transaction processing. The necessity for transaction isolation in databases is implemented via concurrency control techniques. A concurrency control mechanism's success depends on how conflicts are handled. The common methods of concurrency control employed in databases are listed below:

2.7.9.1 Locking

For accessing shared data objects, these concurrency control approaches employ locks to synchronize concurrent transactions. The locking durations[98], conflict resolution

strategies (such as blocking or aborting), deadlock avoidance approaches, etc., of lock-based concurrency control schemes differ. A common concurrency management approach for serializable isolation is two-phase locking (2PL) [99]discuss in the next sections. Easily, a transaction employing 2-Phase Locking must first acquire any locks it need before releasing any locks. Two stages make up the transaction: the growing phase, during which the transaction acquires all locks necessary for the transaction and never releases any locks; and the shrinking phase, during which the transaction releases the acquired locks and never obtains any locks. As a result, 2Phase Locking may need a transaction to wait a long time for a lock under workloads with heavy contention. Since it only permits transaction execution after all conflicts have been resolved through locking, lock-based concurrency control is a pessimistic approach to controlling concurrency. time to lock up. Since it only permits transaction execution after all conflicts have been resolved through locking, lock-based concurrency control is a pessimistic approach to controlling concurrency.

2.7.9.2 Two Phase Locking Protocol

The two-phase commit (2PC) protocol is a commonly used algorithm for ensuring strong consistency in ***NoSQL*** databases. It is a classical algorithm for distributed transaction management that allows all participating nodes to agree on a common transaction coordinator who manages the transaction across all nodes involved in the transaction. Most of the studies may use the 2PL "Two Phase Locking Protocol" as a technique that only accepted method of ensuring Serializability. According to the protocol, a transaction should consist of two phases: an expanding/growing phase and a subsequent contracting/shrinking phase. Locks can be obtained but not released in the first, whereas they can be released but not acquired in the second. The locks are either shared read locks or exclusive write locks. According to the rule, many operations can

hold concurrent read locks on the same object while one transaction has a write-lock, preventing the others from reading or writing to that resource. In addition, it is impossible to obtain a write lock on an item that has a read lock on it.

As **NewSQL** and Relational Database Management databases prefer consistency above availability, they employ the Two-Phase Commit (2PC) protocol, a master-slave based method, to provide distributed consistency. the following two phases:

Phase of Preparing:

1. Each slave receives a Prepare request from the master.

2. The actions are carried out by the slaves, who then notify the master of their prepared answers (either commit or abort). The assets are now secured.

3. The master compiles all of the feedback.

Phase of committing:

1. The master chooses whether or not the transaction is successful and informs all of the slaves of that choice.

2. The slaves finish the operation's commit (or rollback), release the resources, and then acknowledge the master.

3. The master receives every answer and globally commits (or rolls back).

The main concern of 2PL is the performance decrease brought on by lock contention. Long-running reading queries also impede updates as long as writers must get an exclusive lock on a database. In fact, in such a situation, it is conceivable that an

analytical query will get a read-lock on the whole database, halting all concurrent writes for the length of the initial query. For this reason you have fail to apply the 2PL in the ***NoSQL*** database.

The Two-Phase Commit protocol has a weakness in that it is not tolerant to network partitions. The slaves continue to lock the resources if the master fails after the committing request phase since they are unsure of whether they need to commit or rollback.

The issue in this situation is that the slaves are unable to agree on whether or not the transaction is successful.

As we shall explore in the next section, the problem is related to the more general shared consensus problem, and both distributed databases and file systems have adopted a number of alternate solutions.

The protocol works in two phases as follows:

1. In the first phase, the transaction coordinator sends a "prepare" message to all participants to ensure that all nodes can commit the transaction. All nodes respond either with a "yes" or a "no" indicating whether they can commit the transaction or not. If all nodes respond with "yes", the transaction is allowed to proceed to the next phase. However, if one node responds with "no", the transaction is aborted.
2. In the second phase, the transaction coordinator sends a "commit" message to all the nodes indicating that the transaction has been approved for committing. Upon receiving the "commit" message, each node carries out the transaction and then sends an "acknowledge" message back to the transaction coordinator indicating that the transaction has been successfully completed.

If any node fails to acknowledge receipt of the "commit" message, the transaction coordinator sends a "rollback" message to all nodes instructing them to abort the transaction.

The two-phase commit protocol ensures that all nodes agree on whether or not to commit a transaction before updates are made, thereby ensuring strong consistency across all nodes involved in the transaction. However, the two-phase commit protocol is known to suffer from performance issues, especially in large-scale distributed systems where the network latency can significantly impact performance

2.7.9.3 Protocol for Timestamp Ordering (TSO)

With the Time-Stamp Ordering protocol, concurrency may be controlled without blocking. A transaction is given a distinct timestamp at the start of the transaction execution according to timestamp ordering. The timestamps are used by concurrency control to determine the order in which transactions are executed. For instance, if a new transaction Xj enters a serializable system utilizing TSO and a transaction Xi has been given timestamp TS(Xi), then T S(Xi) TS (Xj). The system must make sure that transaction Xi seems to have been done before transaction Xj if both transactions are committed[100].

2.7.9.4 Optimistic Concurrency Control (OCC)

Because it believes conflicts are rare, optimistic concurrency control runs transactions speculatively without requiring the holding of locks for any data items. Nonetheless, conflicts between concurrent transactions might arise during execution. In order to ensure the necessary isolation level, optimistic concurrency control can resolve conflicts at transaction commit time via backward validation. A transaction must be abandoned by OCC if the backward validation fails. Since validation occurs at the end

of execution, when the unsuccessful transaction has already used all the resource (CPU, I/O, network, etc.) of the transaction processing systems, OCC results in high abort rates under conditions of heavy contention.

2.7.9.5 Multi-Version Concurrency Control

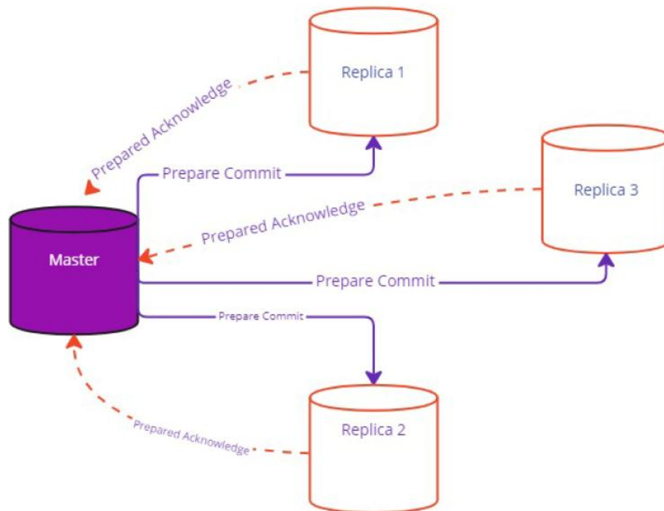
As an option to Serializability, Multi-Version Concurrency Control (MVCC) achieves transactional consistency with snapshot isolation (SI), which has weaker guarantees than Serializability, by using timestamps and incremental transaction ids. The key concept is that an update produces a new version rather than changing the original object. for example when we have a transaction X observes the database state as having been created by all transactions that had already committed before transaction X had began, with no influence from overlapping transactions and no need to lock write or read operations, we can prevent dirty reads, non-repeatable reads, and phantoms. If a concurrent transaction committed an update of a record that transaction X intends to edit, the ***NoSQL*** DBMS aborts transaction X in order to prevent the Lost Update. This principle is also known as "First-Committer-Wins." Nonetheless, abnormalities like Write Skew are conceivable.

M.J. Cahill et al. .[101] have suggested an illustration of such an oddity. Assume we have a database called Responsibilities that records the doctor-working shifts at a hospital. There must always be a doctor on duty for each shift, and they can either be "on duty" or "on reserve." A transaction that wishes to reserve a physician will first update their record, then count the number of physicians on call and terminate the process if none are available. But, in this case with just the *Snapshot Isolation* (SI) , if all doctors on duty attempt to activate "reserve" at the same time, they will be successful since they won't be able to notice the contradictory updates.

While there are currently available methods for serializable isolation for MVCC, they either only apply to in-memory databases that had been suggested by study of [102] or involve keeping track of the whole read set of every transaction, which imposes a significant burden for read-intensive applications[101].

The only timestamp ordering method used by VoltDB[103], which divides the database into parts and schedules transactions to run one at a time at each partition, is a last alternative strategy. The disadvantage of such a strategy is that a transaction that affects several partitions slows down the entire system and causes nodes to remain idle because of network latency.

Phase 1: Preparing Phase



Phase 2: Commuting Phase

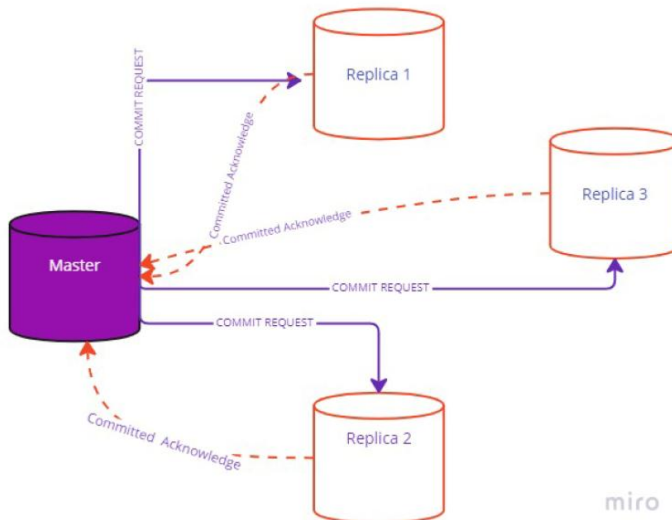


Figure 2.9: Two-Phase Commit Locking Algorithms

2.7.10 Transaction Isolation Levels

The ***NoSQL*** storage systems may simply guarantee consistency and isolation features if all transactions are carried out sequentially. Yet, due to the following factors, the majority of modern systems permit concurrent transaction execution:

1) Enhancing the throughput. Concurrent transactions, for instance, might execute simultaneously on different cores of the same server or on multiple servers, and one transaction may utilize CPU resources while another may experience I/O blocking.

2) Shortening wait times. Think about a workload that consists of both lengthy and short transactions. A small transaction could have to wait until a preceding lengthy transaction is finished if transactions are processed sequentially.

We collectively refer to any methods that maintain speed while ensuring accurate outcomes consistency even when several actions are carried out simultaneously as concurrency control. Yet, there are several variations in the definition of consistency. In addition to the definition of ACID in previous section (0.0) we found that:

Definition 2.1: consistency means the assurance that transactions experience the full consequences of transactions made in the past.

Definition 2.2: consistency means ensures that restrictions on databases, such as table relationships, uniqueness, etc., are not broken.

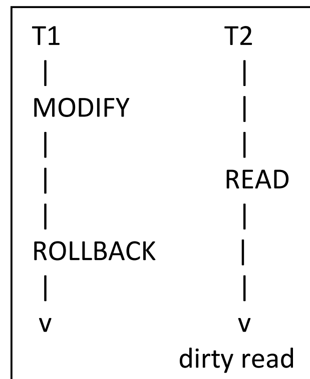
Definition 2.3: consistency means a distributed database's promise states that a finished change is visible to all clients.

The first and last definition probably emphasize the consistency feature of ***NoSQL*** databases. The first definition 2.1 pertains to the idea of serializability, which is the quality that results from concurrent transactions are identical to results from serially processing the transactions without overlapping them.

According to Hal Berenson et al.[104] To guarantee serializability, it is necessary to prevent the following:

1- **Dirty Reads**

A data item D is modified by transaction T_1. The identical data item is then accessed by another transaction, T_2, before T_1 commits or rolls back. In the event that T_1 then does a ROLLBACK, T_2 will have read data that was never committed and so never truly existed.



2- **Dirty Writes**

A data item D is modified by transaction T_1. Before T_1 executes a COMMIT or ROLLBACK, another transaction T_2 makes additional changes to the same data item D. The proper data value is not evident whether T_1 or T_2 then does a ROLLBACK.

3- **Non-Iterate Reads**

Consider the fact that transaction X reads data item "D" in both transactions X and Y. Then, after committing, another transaction Y updates or deletes the data item "D." The value it obtains or learns that the data item has been destroyed if X then tries to reread data item "D"

4- **Lost Updates**

When transaction "A" receives a data item "D," transaction "B" changes the data item "D" (perhaps based on a prior read), transaction "A" updates the data item "D," commits, and the lost update anomaly occurs.

5- Phantoms

Let's say transaction "A" reads a collection of data items "D" that meet certain criteria. After creating data items "D" that meet "A's" search criteria, transaction "B" commits. If "A" then repeats its read under the same circumstances, it receives a different collection of data items than it did during the initial read.

6- Read Skew

Let's say transaction T_1 reads item "x" and commits after updating both item "x" and item "y" to new values. If T_1 reads y now, it could detect an inconsistent state and output an inconsistent state as a result[104].

7- Write Skew

Assume we have two transactions, T_1 and T_2, where T_1 reads x and y and commits them after ensuring they are compatible with a certain constraint. T_2 then reads x and y, writes x, and commits them as well. T_1 then writes y. If x and y were subject to a restriction, it may be broken[104].

2.7.11 Conflicts Read and Write Transactions

Several ***NoSQL*** Databases have abandoned robust transactions in favor of a simplified but more scalable architecture in some mission-critical systems, such shopping carts and social network storage, to close the performance gap between traditional relational databases and web-scale data requirements. Although many of these ***NoSQL*** systems

offer high-level query languages, they only offer a considerably more limited API with poor consistency and only partially support the relational model. These design decisions are influenced by a number of factors, including the requirement for a more flexible schema to support a wider range of data sets, the innate trade-off between consistency and availability during a network partition [24], and the innate trade-off between consistency and latency [6]. While ***NoSQL*** systems are popular in business and receive a lot of academic interest, another school of thought supports the ongoing usage of transactions and high levels of consistency in distributed storage systems [8], The atomic insertion of a comprising up to tens of thousands of changes is supported via write-only transactions by default. For instance, erasing metadata in system X and inserting in system Y are required for atomically shifting a collection of files from system X to system Y. Another use for these transactions is the automated reconfiguration of distributed systems [83] for data migration and dynamic replication factor modification [75, 88].

Read-only or write-only transactions in ***NoSQL*** storage systems are seen as being fundamentally expensive, yet being beneficial in reality. Since read-write and write-write conflicts are likely, concurrency management is necessary for transaction isolation, and distributed commitment protocols are relied upon to guarantee atomicity in the event of failures.

The use of transaction processing technologies is essential for reliable information exploitation and cogent data management. For database systems to accurately reflect the events and activities that occur in the real world, transactions must access or change connected data objects simultaneously. Any stoppage or interleaving of updates and accesses from other transactions throughout each transaction may result in inconsistent data. As the industry standard for database systems, the ACID semantics: Atomicity,

Consistency, Isolation, and Durability, are used to describe transactions. Conflicts arise between concurrent transactions when at least one of them aims to alter a shared piece of data. To resolve the conflicts and provide the necessary transaction isolation levels, concurrency control methods must be used. The consistency and scalability of the systems are significantly influenced by the designs of the concurrency control mechanisms: ***Concurrency control techniques*** (**CC**) keep databases systems in a consistent state by giving an impression of isolated execution, but they can also negatively affect database system performance by preventing the execution of conflicting transactions.

Coordination avoidance has been the subject of recent research [12, 82], as well as the more general trade-off between transaction isolation and performance under the following premise: concurrent serializable transactions under read-write or write-write conflicts require expensive synchronization, which may come at a high performance cost [12]. This presumption, however, ignores the fact that, in the absence of concurrent read-write conflicts, conflicting writes may not always block one another or violate serializability.

Serializable read-only and write-only ***NoSQL*** transactions are used as a counterexample in this thesis to demonstrate that concurrent transactions may be executed in parallel with little overhead even when they conflict.

Although less effective than generic ACID transactions, atomic read-only or write-only transactions have been the subject of heated controversy in recent research [30, 39, 65, 85]. They work effectively in systems that must have atomicity for each batch of reads and writes in order to process reads and writes efficiently. (i.e., two writes within one batch must both succeed or both fail).

Many systems either sacrifice serializability while offering an alternate type of strong consistency [14, 59] or bite the bullet and incur a performance cost for serializable distributed transactions [9, 29, 87]. The transactional solutions rely on expensive atomic commitment methods in that committing a transaction in the presence of contention necessitates at least two network round trips. As contention rises, such protocols' [70, 93] large contention footprint is readily turned into a performance bottleneck, resulting in extra protocol messages as competing transactions settle their relative serialization order. Searching for a read-only/write-only (ro/wo) transaction protocol that is simpler and more streamlined, The system described in this thesis uses many sources to address the issues with read/write conflect. In particular, this thesis suggests a method for enabling read-only and write-only serializable multipartition transactions by dividing time into read-only and write-only periods.

The timestamp concurrency control (TCC) technique used in this architecture uses little data and reduces conflicts across multi-partition operations to achieve high throughput. This study integrates TCC into a scalable distributed key-value store and contrasts it with RAMP [14], a scalable distributed transaction protocol that can only offer a poor isolation level, in order to better understand the performance envelope of TCC.

According to the experimental findings, the suggested model outperforms RAMP in terms of throughput and latency by up to three orders of magnitude when transaction sizes are more than 10 key-value pairs while also offering superior transaction isolation.

2.7.12 Snapshot Isolation Protocol

Snapshot isolation is a concurrency control mechanism in databases that allows transactions to read consistent data as of a specific point in time, without being affected by other concurrent transactions. In ***NoSQL*** databases, which are designed to handle

large-scale distributed systems, snapshot isolation can be implemented using a variety of algorithms.

One common approach is to use vector clocks or version vectors to track the read and write operations on each data item. A vector clock is a list of (node ID, counter) pairs, where each node ID represents a server or process that has performed an operation on the data item, and the counter represents the number of operations performed by that node. When a transaction reads a data item, it records the current vector clock for that item. When the transaction commits, its updates are tagged with a new vector clock that reflects the changes it made.

To ensure snapshot isolation, the database must prevent transactions from reading data that has been modified after they started. One way to do this is to maintain a global ordering of all transactions and their timestamps. When a transaction starts, it is assigned a timestamp that is greater than the largest timestamp used by any previous transaction. Any subsequent transactions that start after the first transaction must have a higher timestamp. When a transaction reads a data item, it checks the item's vector clock to verify that no later transactions have updated that item.

If a conflict is detected, the database can either abort one of the transactions or apply a resolution policy, such as last write wins or merging the conflicting updates. The choice of resolution policy depends on the application requirements and the consistency guarantees provided by the database.

NoSQL snapshot isolation is a type of database isolation level that allows concurrent transactions to read a snapshot of the database at the start of the transaction, ensuring that the data read by the transaction is consistent with the snapshot. This means that the

data read by the transaction will remain the same throughout the transaction, even if other transactions modify the same data in the database.

When all of the interleaved concurrent executions of transactions are similar to serial executions, which is the optimum transaction execution schedule in a distributed ***NoSQL*** database. A schedule like this is said to as serializable. Using two-phase-commit (2PC) is a popular technique for guaranteeing a serializable schedule in a distributed system. Nevertheless, a prior research S. A. Weil et al.[105] demonstrated that this strategy may not scale effectively since one dispersed transaction participant may block while awaiting other transaction participants. In database management systems, Snapshot Isolation (SI) is an isolation level that does not guarantee serializability (DBMS). Yet, because it avoids the majority of frequent concurrency issues and improves the number of concurrent transactions by never letting a read operation stall an update, it is appealing to implement it on distributed databases. The implementation of SI requires to maintain numerous copies of the same data item in addition to this.[104].

As a transaction (Xi) starts in the Snapshot Isolation, it receives a start timestamp (TSi), and when it commits, it receives a commit timestamp (TCi). When reading data item D, Xi always reads the version produced by the most recent transaction of all those that have already committed before TSi. Every time Xi modifies data item D, a new version is produced.

A constraint known as the First-Committer-Wins (FCW) rule is also enforced by Snapshot Isolation. If transaction X2's commit timestamp TC2 falls within transaction X1's life [TS1,TC1], X1 may only successfully commit if X2 did not write data that T1 already wrote; otherwise, T1 would abort.

Snapshot isolation face a problem of anomalies. By interspersing transactions that individually preserve consistency, snapshot isolation is known to allow anomalies that might result in data consistency violations.

Implementing snapshot isolation in ***NoSQL*** databases can be challenging due to the distributed nature of the database, but some ***NoSQL*** databases, such as Apache Cassandra, support snapshot isolation using techniques such as multi-version concurrency control (MVCC) and causally consistent reads.

Snapshot isolation can improve the performance of read-heavy workloads and reduce the occurrence of concurrency issues, such as lost updates and inconsistent reads, in ***NoSQL*** databases.

2.7.13 Serializability

Informally, transactions must appear to take effect in a sequential manner even though they are carried out concurrently. In traditional relational databases, serializability is usually regarded as the ultimate of transaction isolation. Nonetheless, it is generally a greed that the concurrency control overhead for enabling serializability is expensive[106, 107].

The formal definitions of serializability use histories or schedules as instances. which track a transaction's actions, and equivalency, which clarifies the meaning of the phrase "appear to take effect" after a transaction. The correctness qualities that result from different interpretations of equivalence include conflict serializability and view serializability.

When a database uses multi-version storage, serializability receives a somewhat different evaluation. Such a system tracks the precise data item version accessed by each step in a multi-version (MV) history. If transaction Ti receives item x from

transaction Tj, then I = j, or if Tj is the last transaction prior to Ti that writes into any version of x, then the serial of multi-version history is known as a one copy serial. Similar to view-equivalence, equivalence for two MV histories H and H′ is defined. As each version is only written once at most, condition (3) is trivially satisfied. If an MV history is comparable to a one-copy serial MV history, then it is one-copy serializable (1SR), meaning that the system will act as if it only kept one copy of each data item.

2.7.14 PACLEC

The PACELC is design of a system's behavior both when the network functions well and when it doesn't can be used to define it. In the event of a network split, one of the two must be sacrificed: consistency for availability, or vice versa. One must choose between Consistency and low latency answers when there are no communication problems. Databases may prefer one over the other or provide users to select the appropriate action for each circumstance. In any event, this decision affects the application's speed and scalability as well as that of the database.

2.7.15 PAXOS

Paxos is a distributed consensus algorithm used in ***NoSQL*** databases. It is designed to ensure consistency across multiple replicas while allowing for scalability and availability. The main idea behind Paxos is that it requires all participants to agree upon a single value before any action is taken. In addition, each participant has its own view of the state which ensures that no two nodes have different views of the same state. As a result, Paxos makes sure that transactions are correctly executed regardless of node failures or network partitions. Furthermore, since Paxos does not require messages to be delivered between nodes, it offers better latency than other systems. All these

features make Paxos one of the most popular solutions for distributed consensus problems.

NoSQL transactions consistency remains a significant challenge in the world of distributed databases. Balancing scalability and performance with data consistency is a complex task. Developers must carefully evaluate their application's requirements and choose the right ***NoSQL*** database and consistency model to ensure their data remains both available and reliable in a distributed environment. As the field continues to evolve, new solutions and best practices will emerge to address these challenges

2.8 Previous Studies

In this chapter, some of the studies conducted on *NoSQL* databases will be reviewed, and they varied in their directions, some of which tested the performance element and some the scalability elements.

The first study conducted by Bansal, Neha et al. [29]examine that if the schema does matter, what is its potential impact on application performance? It arises the question of whether the schema matters in ***NoSQL*** databases, given that explicit schema declaration is not required before data storage. Acing three types of ***NoSQL*** databases: Document store, Column store, and Key-Value store.

A study by Liu et al.[82*] proposes a new consistency model for ***NoSQL*** databases called "causal order consistency". Causal order consistency guarantees that all transactions will be applied in the same order across all replicas, even if the transactions are executed by different clients. This can help to improve the consistency of ***NoSQL*** transactions, even in the presence of network partitions or other failures.

A study by Zhang et al.[83*] proposes a new algorithm for mitigating the impact of data inconsistencies in ***NoSQL*** databases. The algorithm uses a conflict detection and resolution mechanism to identify and resolve data inconsistencies in a timely manner. This can help to improve the performance and reliability of ***NoSQL*** applications.

Singh et al.[84*] proposes a new consistency model for ***NoSQL*** databases called "optimistic consistency". Optimistic consistency guarantees that all transactions will eventually be committed, but it allows for temporary inconsistencies to occur. This can improve the performance of transactions, especially in applications with a high volume of concurrent transactions.

A study by Zhang et al.[0] proposes a new algorithm for resolving conflicts in ***NoSQL*** databases. The algorithm uses a distributed timestamp ordering protocol to ensure that conflicts are resolved in a consistent manner. This can help to improve the consistency of ***NoSQL*** transactions, even in the presence of network partitions or other failures.

These studies suggest that there is ongoing research into new ways to improve the consistency of ***NoSQL*** transactions. As ***NoSQL*** databases become more popular, it is likely that we will see even more research in this area in the future.

Study of Noudoust et al. [30] Aimed to improving data consistency, the proposed approach also satisfies the accepted compromise between the pillars of the CAP and PACELC theorems and proposed approach is able to set different levels of data consistency in key-value ***NoSQL*** databases. An approach is based on the parameters set in the ***NoSQL*** -dependent structure and is applied to distributed systems, using the quorum algorithm to adjust the consistency and to determine the different grades of data, and different data writing and reading and mix reading/writing operations are implemented test-bed to study the performance of the proposed approach.

A. Karpenko [31] et al. study the evaluation of performance of distributed fault tolerant computer systems and replicated ***NoSQL*** databases and studying the impact of data consistency on performance and throughput on the example of a 3 replicated Cassandra cluster. This study propose a new method of minimizing Cassandra response time while ensuring strong data consistency which is based on optimization of consistency settings depending on the current workload and the proportion between read and write operations.

Study of Sidi Mohamed Beillahi et al. [32] was investigated the problem of robustness that problem of checking whether a program has the same set of behaviors when

replacing a consistency model with a weaker one. This study was focused on consistency models which are weaker than standard serializability, namely, causal consistency, prefix consistency, and snapshot isolation.

Study of Adam Krechowicz et al. [33] proposed a novel data storage architecture that supports strong consistency without losing scalability It provides strong consistency according to the following requirements: high scalability, high availability, and high throughput based on the Scalable Distributed Two–Layer Data Store which has proven to be a very efficient ***NoSQL*** system. The proposed architecture takes into account the concurrent execution of operations and unfinished operations and test the proposed model into two type of ***NoSQL*** database, MongoDB and MemCache. The finding results of this study show that the proposed architecture presents a very high performance in comparison to existing ***NoSQL*** systems

Study of Gorbenko et al. [34] investigate the relationship between the performance (response time) and consistency setting in Column ***NoSQL*** database and select the Cassandra and report the result of experiments from read and write performance benchmark as a main factors. The researchers deploy the experiment of replicated cluster in cloud service from Amazon EC2. The result of this study found that the strong consistency costs up to 25% of performance within the best settings for the strong consistency are depended on the ratio of read and write operations.

The study of Chenggang Wu et al. [35] is discuss the setting of serverless Function-as-a-Service (FaaS) platforms that support cloud computing which may run on a separate machine and access remote storage and facing how can improving Input/Output latency in this setting while also providing application-wide consistency and proposed A single application may execute multiple functions across different nodes. And present a new

protocols for MTCC (Multisite Transactional Causal Consistency (MTCC)) implemented in a system called HYDROCACHE. The result evaluation of this study is to demonstrates orders-of-magnitude performance improvements due to caching, while also protecting against consistency anomalies that otherwise arise frequently.

Study of Ranadeep et al. [36] draw a question about how the modern databases provide different consistency models for transactions corresponding to different tradeoffs between consistency and availability. and investigate the problem of checking whether a given execution of a transactional database adheres to some consistency models like read committed, read atomic, and causal consistency are polynomial-time checkable while prefix consistency and snapshot isolation are NP-complete in general the findings of this study is devise algorithms that are polynomial time assuming that certain parameters in the input executions, e.g., the number of sessions, are fixed. We evaluate the scalability of these algorithms in the context of several production databases.

Study of Shale Xiong et al.[37] proposed an operational semantics to describe the client-observable behavior of atomic transactions on distributed key-value stores. The proposed model by this study provide operational definitions of consistency models for our key-value stores which are shown to be equivalent to the well-known declarative definitions of consistency model for execution graphs. The study have a specific protocols of geo-replicated databases and partitioned databases can be shown to be correct for a specific consistency model by embedding them in our centralized semantics.

Study of María Teresa González-Aparicio, et al. [38] proposed a various transaction models and protocols and investigate into the testing of transactional services in ***NoSQL*** databases in order to test and analyse the data consistency by taking into account the

characteristics of ***NoSQL*** databases such as efficiency, velocity. This study can assist ***NoSQL*** application developers in choosing between transactional and non-transactional services based on their requirements of the level of data consistency.

Transactional services aim to ensure data consistency by taking into account the characteristics of ***NoSQL*** databases such as efficiency, velocity, etc. the experimental study of this paper was used a Key-value ***NoSQL*** database and test with carried out by using Raik database.

Study of R. Jiménez-Peris [39] describe the generic ultra-scalable transactional management layer and focus on its API and how it can be integrated in different ways with different data stores and databases. And discuss the problem of lies in that when a business action requires to update the data, the data reside in different data stores, and they are subject to inconsistencies in the event of failure and/or concurrent access.

Study of Nazim Faour [40] (2018) a model of simulation to measure the consistency of the data and to detect the data consistency violations in simulated network partition settings. So workloads are needed with the set of users who make requests and then put the results for analysis. As result of this paper is Simulations can only work as an estimation or explanation vehicle for observed behavior.

A study by Wang et al.[86*] examined the eventual consistency model, which offers the weakest consistency guarantee of all. The study found that eventual consistency can be implemented in ***NoSQL*** databases with a very low overhead, but that it can lead to data inconsistencies in some cases.

Study of Huang et al.[41] using a Document ***NoSQL*** database "CASSANDRA" to test the consistency The consistency can be improved by tuning system configurations when users use partial quorum settings. By using the session model of consistency to

analyze the root cause of consistency violation, testifying that the length of the write queue is a reasonable indicator for consistency quantification and •Consistency Of a ***NoSQL*** System. The study recommended configurations by changing the write thread number and the fine-grained quorum setting for enhanced consistency control. Because consistency anomalies do not occur uniformly, we discuss how to stabilize the consistency by analyzing system logs.

Study of González-Aparicio et al. [42] discuss the ***NoSQL*** with side of do not enforce or require strong data consistency nor do they support transactions investigates into the transaction processing in consistency-aware applications hosted on MongoDB and Riak which are two representatives of Document and Key-Value ***NoSQL*** databases and develops new transaction schemes in order to provide ***NoSQL*** databases with transactional facilities as well as to analyze the effects of transactions on data consistency and efficiency in user's applications. And evaluate the proposed schemes by using YCSB +T Benchmark in the experiment. The results shows that Strong consistency can be achieved in MongoDB and Riak without severely affecting their efficiency.

A study by Goyal et al.[0] examined the causal consistency model, which offers a weaker consistency guarantee than ACID but is still sufficient for many applications. The study found that causal consistency can be implemented in ***NoSQL*** databases with a reasonable overhead.

Study of Burdakov [43] was represented a ***NoSQL*** Replication Problems in ***NoSQL*** databases by analyzes the influence of the N, W, R replication parameters on the consistency characteristics of database record replicas (N -- the total number of one record's replicas, W -- number of replicas for write operation execution into a database,

R -- number of replicas for record read operation execution from a database). It describes a developed model for eventual consistency (W+R $\leq$ N), obtaining probability estimate that during the process of N-W replica updates there will be at least one read request out of non-updated replicas It also proposes a model for strong consistency of the replicas in ***NoSQL*** databases, which allows for estimation of random wait time of the read request for the record update completion. It describes the process for preparation and execution of experiments in the cloud for model calibration and its validation. The result of this study found that A model for strong consistency of the replicas in ***NoSQL*** databases allows for estimation of random wait time of the read request for the record update completion.

Study of Adewole Ogunyadeka et al. [44] proposed a new multi-key transactional model is that provide a ***NoSQL*** systems with standard transaction support and stronger data consistency, which is configurable based on application requirements, and preliminary experiments show that it maintains good performance by using document database (MongoDB).

Study of Ayman E. Lotfy et al.[45] was aimed to offer the consistency and ACID features to ***NoSQL*** Databases and given the push to this features vs. relational databases by proposed A middle layer solution using a four phase commit protocol ensures data consistency and ACID properties for ***NoSQL*** databases, allowing for executing many transactions related to each other through updating the same data while increasing scalability and throughput without affecting system availability.

Study of Ogunyadeka [46] proposed a novel Multi-Key transaction model has been designed to ensure stronger consistency and integrity of data, which has been validated

through a prototype system and experiments, showing that it maintains stronger consistency of cloud data as well as appropriate level of reliability and performance.

The study of Madhavamuniappan [47] discuss the transaction processing system in two types of ***NoSQL*** databases that sacrifice of data consistency and the latency of the write operation and response time for web application. The finding of this study is Transaction Processing system allows cloud database services to execute the ACID transactions of web applications, even in the presence of server failures and network partitions.

A study by Chen et al. examined the consistency guarantees offered by a variety of ***NoSQL*** databases. The study found that most ***NoSQL*** databases do not offer ACID transactions across multiple documents, and that the consistency guarantees that are offered vary from database to database.

Islam Md. A and Vrbsky S V. [48] illustrated the benefits and drawbacks of a few consistency strategies. In this article, no approach for performance improvement was introduced.

Wada H, et al. [49] studied what customers notice about the consistency and performance attributes of different products. The researchers have provided no information on the optimization process of various techniques. The gaps in the present study work persuade us to contribute more on the performance of consistency techniques on ***NoSQL***, as well as potential improvements.

The Study of Kraska T, et al. [50] established a new transaction paradigm for ensuring consistency, however nowhere in the study do they demonstrate the sufficient level of performance of alternative consistency approaches such as BASE and Quorum.

Study of A.Dey [51] YCSB+T employs a transaction protocol that is coordinated by the client , which operates across data stores of varying types and relies on the data store's capability of providing strong consistency at the level of individual items. Additionally, the protocol incorporates the ability to add user-defined data and allows for global read-only access. By incorporating these features, the need for a central coordinating system to facilitate transactions involving multiple items while still adhering to the ACID properties is eliminated. The transaction protocol is divided into two distinct phases, each serving a specific purpose. In the first phase, the data items are retrieved from their respective data stores, and subsequently, they are assigned a timestamp in the form of metadata. The second phase is dedicated to the actual commit of the transaction and the updating of a global variable known as TSR, which plays a crucial role in determining the outcome of the transactions. The concurrency among multiple transactions is assumed to be effectively managed by leveraging the test-and-set capability inherent in each individual data store. This capability allows for efficient handling of concurrent access to shared data and ensures that conflicts are resolved in a controlled manner.

In their 2015 paper, Lee et al. proposed a novel solution called RIFL (Reusable Infrastructure for Linearizability) to address the problem of consistency in database system design. RIFL is built upon the concept of remote procedure calls (RPCs) with at-least-once semantics, which means that invocations are retried until successful. However, RIFL takes these semantics a step further by enhancing them to achieve exactly once semantics, which are necessary for guaranteeing linearizability. This is accomplished by assigning a unique identifier to each request and using a persistent log to ensure that completed requests are not re-executed [2].

One of the key findings of the authors' research is that the write overhead of implementing RIFL in RAMCloud is only 4% compared to the base system without RIFL. This demonstrates the efficiency and practicality of their approach. Additionally, the introduction of exactly once semantics simplifies the implementation of transactions, making it easier to manage and coordinate multiple operations within the database system [1].

The foundation of RIFL lies in Sinfonia, an in-memory service infrastructure that provides a mini-transaction primitive for atomic cross-node memory access. By building upon this existing infrastructure, the authors are able to leverage its capabilities and extend them to achieve their goals with RIFL. This approach not only saves time and effort in designing a new system from scratch but also ensures compatibility and interoperability with existing systems.

However, it is important to note that RIFL has a central limitation in its assumption that clients are reliable and do not lose their state upon crashes. This means that if a client crashes and loses its state, RIFL may not be able to guarantee the consistency and correctness of its operations. While this limitation is acknowledged by the authors, it is an area that could be further explored and addressed in future research [2].

In conclusion, Lee et al.'s proposal of RIFL offers a promising solution to the problem of consistency in database system design. With its enhanced exactly once semantics and efficient implementation overhead, it provides a practical approach to achieving linearizability. By building on existing infrastructure like Sinfonia, the authors are able to leverage the strengths of their foundation and extend its capabilities. However, the assumption of reliable clients poses a limitation that needs to be taken into consideration.

Overall, RIFL presents a valuable contribution to the field of database systems and opens up avenues for further research and development.

Chapter Three

Methodology and Conceptual Design

This chapter contains the methodology to test a consistency model vs. performance and in some of ***NoSQL*** databases and evaluate the best model of consistency how can work in this types of database

3.1 Research Methodology

An outcomes monument is thought to include knowledge about advanced tools, techniques, and algorithms as well as presumptions about the environment in which the artifact is supposed to operate. The research's results must either address a problem that hasn't been addressed before or offer a superior solution in order to be regarded as a novel contribution. In this thesis is the most effective strategy for achieving the study's goals, which are outlined in above paragraphs. As a result, we support the recommendation. The processes that make up our adopted framework are shown in Figure 3.1..

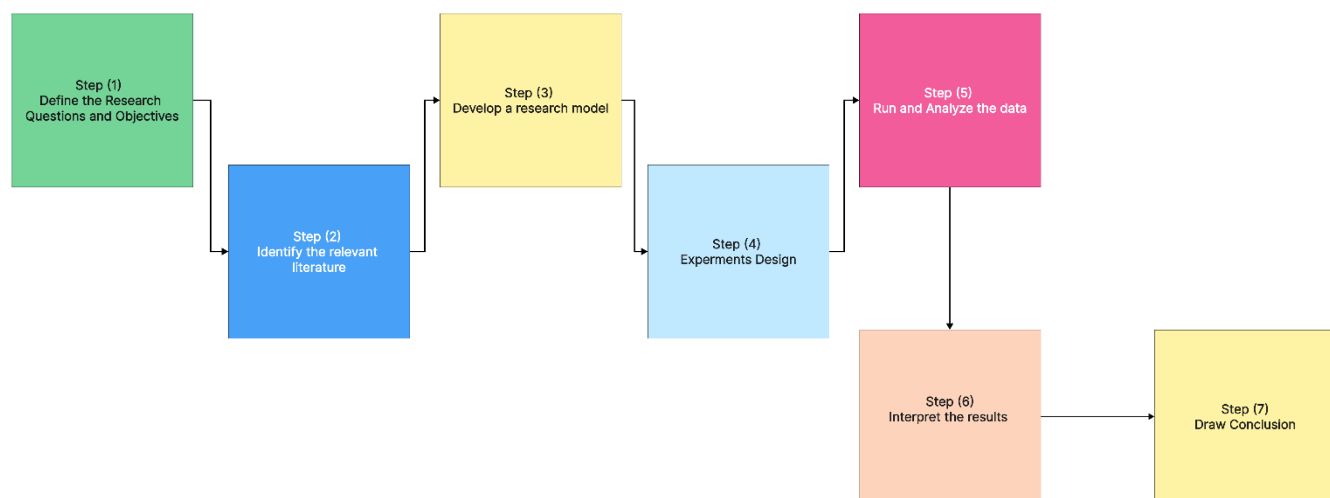


Figure 3.1 shows the application of the research technique's process model.

The research methodology steps for conducting the study:

1- Define the research question.

The first step is to define the research question. This will help to focus in research and avoid getting sidetracked.

2- Identify the relevant literature.

The next step is to identify the relevant literature. This will help you to get a well-rounded view of the topic. There are many different sources of literature, including academic journals, books, and online articles. When identifying the relevant literature, it is important to be critical of the sources you use. Not all sources are created equal. Be sure to evaluate the credibility of the sources you use.

What has been written about ***NoSQL*** transaction consistency?

What are the different approaches to achieving transaction consistency in ***NoSQL*** databases?

3- **Develop a research model**.

The third step is to develop a research model. This is a conceptual framework that will help you to organize your research and guide your analysis. The research model should be based on the research question and the relevant literature.

Consistency model → Performance

→ Scalability

4- Design Experiments and Implementation.

The fourth step is to collect data. This may involve running experiments, surveying users, or analyzing existing data. When collecting data, it is important

to be clear about the data you need to collect. The data you collect should be relevant to the research question and the research model.

5- **Run and analyze the data.**

The fifth step is to analyze the data. This involves using statistical methods to identify patterns in the data. When analyzing the data, it is important to be careful not to over interpret the data. The data should be interpreted in the context of the research question and the research model. This involves using statistical methods to identify patterns in the data.

6- **Interpret the results.**

The sixth step is to interpret the results. This involves drawing conclusions about the research question based on the data you have collected and analyzed. When interpreting the results, it is important to be clear about the limitations of the study. The results of the study should be interpreted in the context of the research question, the research model, and the limitations of the study.

7- **Draw conclusions.**

The seventh step is to draw conclusions. This involves summarizing the findings of the study and discussing the implications of the findings for the design and use of ***NoSQL*** databases. When drawing conclusions, it is important to be clear about the limitations of the study. The conclusions of the study should be based on the findings of the study, the research question, the research model, and the limitations of the study.

- To discover the performance factors that causes inconsistency of the data in *NoSQL* databases, meta-Analysis method, which involves analysis and combination of multiple scientific studies, has been followed. Related literatures were systematically reviewed, and then synthesized the results. However, academically authoritative articles, journals, course literature or recommended books, research materials, text on trusted online site were prioritized for most literature reviews. Database such as ACM digital library, DiVA portal were used as a leading source to identify the related research work. All the key aspects of systematic literature. The review was conducted to extract data comprehensively that relates the first research problem.
- Defining the proposed model.
- The researcher divided the experiments into two stages:
    - The first stage: a pre-test to choose which models of consistency are suitable to work on
    - The second stage: A post-experiment to test the accuracy of the proposed model for transaction consistency in all types of *NoSQL* databases

3.2 NoSQL Models Experiment

- This study focuses on the best approaches used to be analyzed; by comparing with multiple approaches in the popular type of ***NoSQL*** databases. The analysis includes reading and examining articles and documents on those approaches to identify the process and outcomes. Furthermore, the satisfactory settings of the consistency performance was realized through the practical experiment conducted using YCSB benchmarking tool. The details of YCSB system architecture is presented in next section of tools 4.2.

- In order to show the performance, the researcher chosen consistency parameters are *Read*, *Insert*, and *Update*. *Read* reads a record from the database. It can read both a randomly chosen field and all the fields. Insert inserts a new data. *Update* updates a record by replacing the value from a field. The metrics for consistency we chose were latency (ms) and throughput (ops/sec). The latency for each parameter (e.g., read latency, update latency) and the amount of throughput completed for each execution were used to judge the consistency level for each database.
- The researcher select the performance metrics is measured by the metrics latency verses throughput. That means when with a given throughput, the less latency occurred, the better performance is. Thus more reliable and consistent data achieved. The benchmarking tool, YCSB shows the latency against any given throughput (ops/sec), which is useful to measure the satisfactory settings of the respective ***NoSQL*** database system.
- The suggestion to improve one of the approaches was decided based on the satisfactory level of their consistency. Solving this question involved experiments to identify issues with Quorum. To do that, experiments will be conducted with the latency of Scylla database system by different workloads (in the same tool YCSB) using different read-update ratios.
- Variations will be introduced by creating different threads to justify the user experiences of updating data simultaneously. At the end of those tests, it is expected to get a clear picture of the performance of Quorum in various use cases. These results help to decide whether and in which cases, Quorum has better performance regarding consistency.

3.2.1 Pre-Experiment Tools

YCSB: Database Benchmarking Tool:

With a growing list of new databases, such as MongoDB, Azure, Cassandra, Scylla and many others, determining which ***NoSQL*** database is appropriate for any distributed application is difficult, partly because the features of each database differ in some way, and there is no divine way to measure the consistency level of one database versus another. This requirement compelled the researchers to create YCSB, which provides a standard set of workloads for analyzing the behavior and performance of various cloud database systems. YCSB is a common benchmarking tool for evaluating various cloud systems and ***NoSQL*** databases. YCSB is a common benchmarking tool for evaluating various cloud systems and ***NoSQL*** databases. The YCSB project was created by a group of computer scientists from Yahoo Inc's research division. It was created in Java and contains functionality that allows users to leverage the current API to benchmark their own database. YCSB does not provide a graphical user interface for interacting with its users. The framework's most recent release (version 0.10.0) is available on Github, where users can clone or download the source file to their desktop. The YCSB framework is divided into two parts: the client, which is an extendable workload generator, and the core workloads, which are a collection of workload scenarios that will be run by the generator[67].

3.2.2 YCSB Workloads

YCSB contained a six core workloads. In many cases, the fundamental workloads are sufficient to evaluate a system's performance and consistency settings. Three distinct sorts of workloads were chosen for this study: Workload A, Workload B, and Workload D, each with a different consistency parameter. Workload C was not chosen since its read operation is already included in other workloads.

Table 3.1 YCSB Workloads

Workload	Operations	
A-Update Heavy	Read: 50%	Update: 50%
B-Read Heavy	Read: 95%	Update: 05%
C-Read Only	Read: 100%	Update: 00%
D-Read Latest	Read: 95%	Insert: 05%
E-Short Ranges	Scan: 95%	Insert: 05%

With this in thoughts, and also to avoid unnecessary technological difficulty, we did not run a scan operation in any of the databases, which means Workload E was also deleted from the list of options. After all, all workloads were chosen in such a way that they are compatible with typical current applications.

However, the basic workload does not cover all system characteristics and database situations, which is why we need to expand the client when we need to design other workloads. This functionality allows developers to run any ***NoSQL*** database and obtain benchmarking results. YCSB now supports 19 distinct ***NoSQL*** databases.

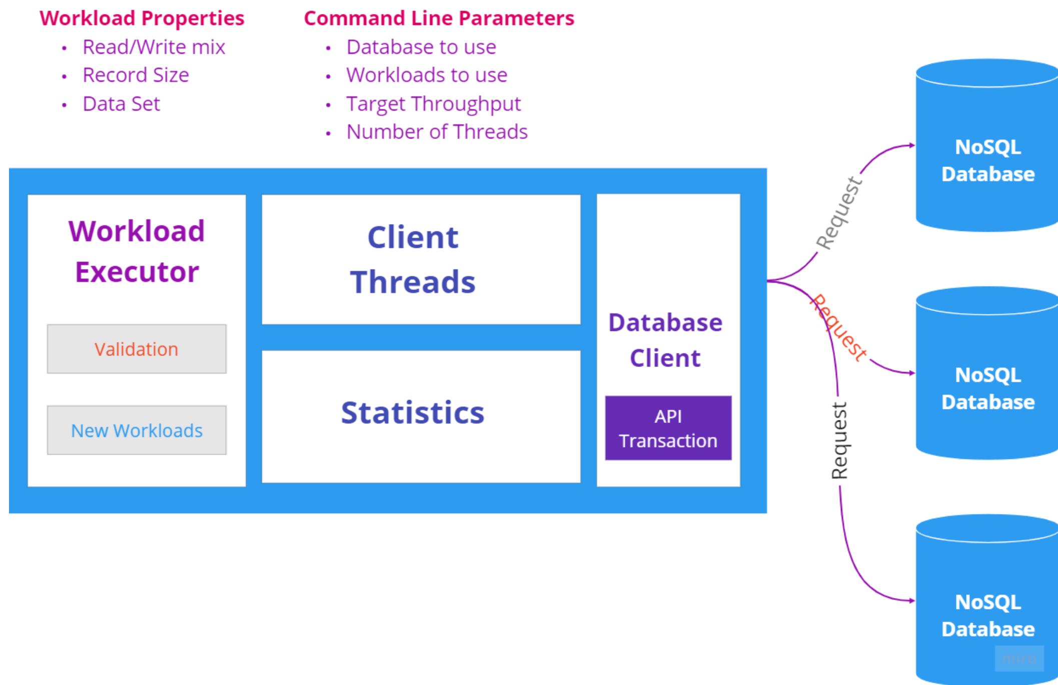


Figure 3.2 YCSB Architecture

Several manipulations, such as building project modules, implementing java methods, and running database queries, are required in the main project in order to execute any experiment with other databases. However, because the databases to be evaluated are already packed with the YCSB, no further database interface layer is deemed to be developed. Figure 4.1 demonstrates the architecture components of the YCSB client. The client framework performs two basic functions: it generates data that must be fed into the database and it generates the operations that eventually compose the workload. Multiple client threads are executed by the workload executor. Each thread performs a sequence of activities sequentially by calling the database interface layer to load the database and execute the job.

Furthermore, each client thread evaluates delay and so achieves operation throughput. Client's operation is defined by a set of

characteristics such as name/value pairs. By convention, these operations are classified into two types:

1. ***Workload properties***, which are used to specify the workload. For example, the read-write mix, use distribution, and the number and size of database fields.
2. ***Runtime properties,*** specify how properties are specified for a given experiment.

For example, the number of threads and the database to be used, among other things.

Furthermore, the following steps were taken to run the workload against each database.

3.2.2 Pre-Experiment Overview

This experiment aims to assess the performance of data consistency in a ***NoSQL*** database system under various scenarios. Specifically, we will evaluate the impact of different consistency levels on read and write operations in the database. The experiment will use a synthetic dataset and measure latency, throughput, and data integrity to draw conclusions about the trade-offs between consistency levels.

3.2.3 Pre-Experimental Steps:

1. **Select *NoSQL* Database:**

We used a popular ***NoSQL*** database system, with built-in support for different consistency levels. The researchers use a column ***NoSQL*** database named ScyllaDB. ScyllaDB is a distributed ***NoSQL*** wide-column database for data-intensive applications that require high performance and low latency, its shared

cluster, replica set or standalone, It is an open source ***NoSQL*** database and support cloud[68].

2. **Dataset:**

   A synthetic dataset of 1 million records with varying data structures and sizes, designed to simulate real-world use cases.

3. **Consistency Levels:**

   A restricted number of alternatives have been offered in order to achieve consistency. As previously stated, the shortcoming of these various techniques is that they do not preserve complete consistency. Database performance difficulties are becoming increasingly widespread as the volume of data processed by organizations' information systems grows. Meanwhile, customer requirements and expectations are continually increasing, and a delay in response time might have a significant impact on the operations[69].

   As a result, a significant amount of work should be spent measuring the satisfactory level of performance of currently employed methodologies. Not to add, methods such as Quorum and eventual consistency are two of the most commonly employed properties in many prominent ***NoSQL*** database solutions. Some notable ***NoSQL*** databases, such as MongoDB, Dynamo, BigTable, Cassandra, and ScyllaDB, adhere to the eventual consistency principle.[70, 71]. Quorum, on the other hand, is a classic approach that has long been utilized in ***NoSQL*** storage systems. Because the system is symmetric, Quorum has grown in favor among developers. Symmetry means that the quorum system is consistent and fair (i.e., all quorums have the same size and burden is spread uniformly among nodes in regions)[72]. As a result, we chose Quorum and

eventual consistency to be researched in this study since they are both inevitably prominent approaches utilized in current distributed systems.

In this study the researchers tested two different consistency levels:

- Eventual Consistency
- Quorum Consistency

3.2.4 Pre-Experiment Benchmark

The Research use a benchmark tool that makes use of the features of various workloads in order to run the experiment and assess the consistency levels, we use the Yahoo Cloud Service Benchmark (YCSB) 1.12.0. YCSB can be utilized with a variety of programs, including Additionally, YCSB displays genuine cloud features like scale-out, elasticity, and high availability, and we use it to execute Workload A, a workload with a high read-to-update ratio (60:40). After replication, our workload in both environments consists of 10 million operations on 5 million rows for a total of 50.84 GB of data.

com.yahoo.ycsb.BasicDB, a java class, is provided by the YCSB Client as a basic fake interface layer. It converts read, insert, update, delete, and scan calls made by the YCSB Client into database API calls. The class name was supplied on the command line, and the YCSB client loaded it. Any command-line attributes or parameters can be provided to the interface instance. To setup the interface layer, for example, we gave the hostname of the database we benchmarked .However, in order to load the entry into the database, the right workload has to be specified. The workloads are intended to manage two phases: the loading phase, in which data is entered into the database, and the

transaction phase, in which operations (such as read and update) are performed on the inserted record .

3.2.5 Pre-Experiment Workloads

The workload type must be selected before loading any number of records into the table. The throughput (ops/sec) and latencies vary depending on the workload. The YCSB client inserts 1000 entries by default. However, with the appropriate command option, a custom number of records to be inserted (i.e., record_count=100000) can be specified. When the loading phase gets the instruction identifying the local host and port number on which the database cluster is executing, it will begin inserting records. The following Windows shell command loads data into the Cassandra database.

binycsb load scylladb-cql -P workloads/workloada -p hosts=127.0.0.1 -p port=9042 "C:\Python\python.exe"

The output also includes other information such as cluster and datacenter. The benchmark's final stage (i.e., transaction phase) is to run the workload. The command to run the workload differs somewhat from the previous one: binycsb run scylladb-cql -P workloads/workloada -p hosts=127.0.0.1 -p port=9042 "C:\Python\python.exe"

Workload Operations

- Read Operations: Randomized read requests on the dataset.

- Write Operations: Randomized write requests on the dataset.

```
root@myvps986397:~/ycsb-0.17.0# ./bin/ycsb run scylladb -s -P workloads/workloada -p scylladb.url=scylladb://localhost:27017/ycsb?w=0
----------------------------------------------------------------------------------------------------------------------------------------
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -cp /root/ycsb-0.17.0/scylladb-binding/conf:/root/ycsb-0.17.0/conf:/root/ycsb-0.17.0/lib/htrace-core4-4.1.0-incubating.jar:/root/ycsb-0.17.0/lib/core-0.17.0.jar:/root/ycsb-0.17.0/lib/jack
Command line: -db site.ycsb.db.scylladbClient -s -P workloads/workloada -p scylladb.url=scylladb://localhost:27017/ycsb?w=0 -t
YCSB Client 0.17.0

Loading workload...
Starting test.
2022-11-26 06:21:46:160 0 sec: 0 operations; est completion in 0 second
mongo client connection created with scylladb://localhost:27017/ycsb?w=0
DBWrapper: report latency for each error is false and specific error codes to track for latency are: []
2022-11-26 06:21:47:814 1 sec: 998 operations; 562.89 current ops/sec; [READ: Count=516, Max=137471, Min=312, Avg=1094.47, 90=1185, 99=5487, 99.9=15247, 99.99=137471] [CLEANUP: Count=1, Max=11279, Min=11272, Avg=11276, 90=11279, 99
[OVERALL], RunTime(ms), 1784
[OVERALL], Throughput(ops/sec), 559.4170403587444
[TOTAL_GCS_G1_Young_Generation], Count, 4
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 31
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 1.7376681614349778
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 4
[TOTAL_GC_TIME], Time(ms), 31
[TOTAL_GC_TIME_%], Time(%), 1.7376681614349778
[READ], Operations, 516
[READ], AverageLatency(us), 1094.4728682170542
[READ], MinLatency(us), 312
[READ], MaxLatency(us), 137471
[READ], 95thPercentileLatency(us), 2469
[READ], 99thPercentileLatency(us), 5487
[READ], Return=OK, 516
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 11276.0
[CLEANUP], MinLatency(us), 11272
[CLEANUP], MaxLatency(us), 11279
[CLEANUP], 95thPercentileLatency(us), 11279
[CLEANUP], 99thPercentileLatency(us), 11279
[UPDATE], Operations, 482
[UPDATE], AverageLatency(us), 614.9232365145228
[UPDATE], MinLatency(us), 157
[UPDATE], MaxLatency(us), 9743
[UPDATE], 95thPercentileLatency(us), 1248
[UPDATE], 99thPercentileLatency(us), 6751
[UPDATE], Return=OK, 482
----------------------------------------------------------------------------------------------------------------------------------------
```

Figure 3.3 Terminal output of loading phase

Replication factor:

The replication factor is setting equal 3 replica.

3.2.6 Pre-Experimental Metrics

- Latency: Measured in milliseconds, indicating the time taken for a read or write operation to complete.
- Throughput: Measured in operations per second (OPS), representing the number of successful operations completed per second.
- Data Integrity: Ensured by comparing the retrieved data with the expected data.

3.2.7 Pre-Experimental setup

We deploy a single replica set in Amazon Elastic Compute Cloud (AWS) to conduct the tests (EC2). With 30 nodes on the USA (us-east-1) site and 5 nodes in the same geographical region and different availability zones, we deployed ScyllaDB on two data zones. Each node has the following specifications:

- **250 GB NVMe SSD,**

- **32 GB of Memory,**
- **8-cores INTEL CORE.**
- **Standard architecture of 1000 Gbit/s dark fibers**
- **OS: Linux Ubuntu 18.4**

With ScyllaDB, we used a replication factor of three copies, with two of them allocated to Zones 1 and 5.

3.2.8 Pre-Experiment Results

The reasons and motives for data discrepancy are discussed in the following subsections. The following elements that create data discrepancy are discovered by reviewing and assessing accessible scholarly literature in the given issue area.

Workload Throughput: Test 1:

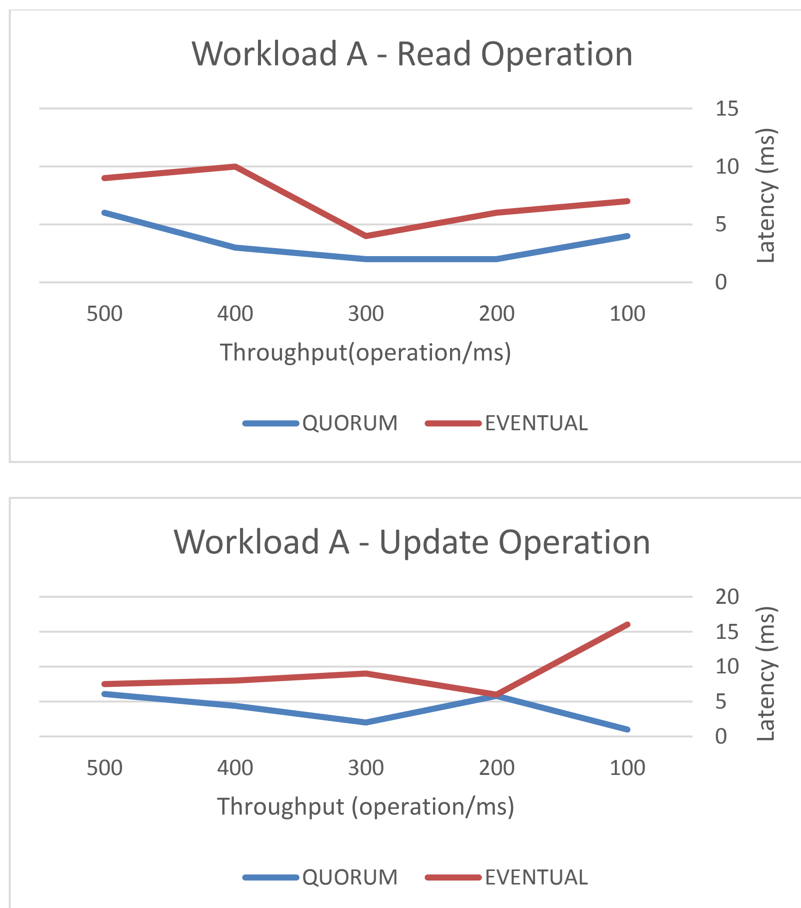


Figure 3.4 workload A – Throughputs

The delay vs. throughput curves for each read and update operation for both eventual consistency and Quorum Consistency are shown in Figure 3.3. The graph shows that operation delay increases as throughput grows for both consistency models. For the read operation, quorum consistency obtained lower latency than eventual consistency, despite the fact that the rising rate for both systems per throughput is quite sluggish. In quorum consistency, the highest throughput (i.e., 10000 ops/sec) differs from the starting throughput (i.e., 516 ops/sec) by only 4 ms, whereas it is 5 ms. The system's performance varies greatly during the update procedure. Quorum displays an unusual increase in latency for every 1000 operations per second to update.

Workload Throughput: Test 2:

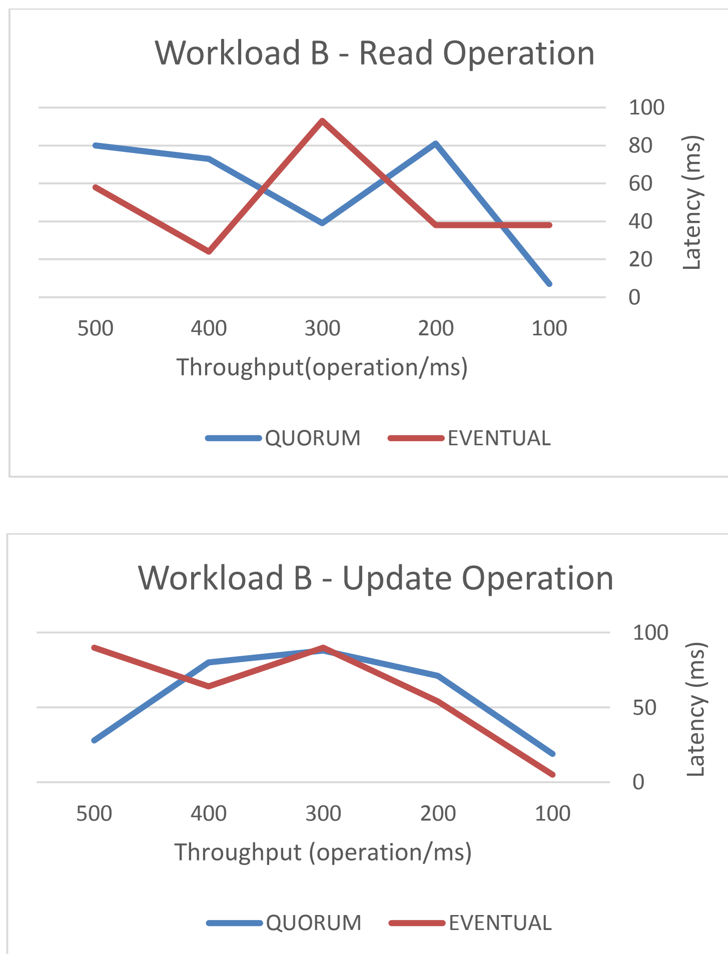


Figure 3.5 workload B – Throughputs

Figure 4.4 shows that workload B test information's. At large throughputs, quorum has decreased read latency for each operation. However, in this test, both systems' operating delay did not rise as throughput increased. Quorum reacted negatively in comparison to eventual, demonstrating that for quorum, latency reduces to the lowest for the largest throughput, but for eventual, latency increases as throughput increases. Both data stores have considerably higher read latency. In quorum, latency for update operations is remarkably low. When the throughput reached 1000 ops/sec, the latency dropped to 1 ms and stayed constant while the throughput increased by 500 ops/sec. In the end, both tasks performed consistently while increasing delay in accordance with throughput.

Workload Throughput: Test 3:

Figure3.6 workload D – Throughputs

Figure 3.5 shows the results of insert and read operations. The ultimate consistency model fared shockingly poorly while adding records in this final test. Inserting data for each of the throughput amounts took more than 18 ms. Quorum, on the other hand, excelled at read operation. No delay was computed until it reached 954 ops/sec throughput, and this increase was gradual, as 1 ms is the minimum latency evaluated. Then, as time passed, consistency brought comparable results to its insert action. As a result, it is clear that it's read and insert operations differ significantly from one another.

According to the test findings, quorum took much less time than eventual to read, insert, and update values from one node to another. Figures 3.3, 3.4, and 3.5 indicate that quorum exceeded eventual when mapping delay vs throughput with various workloads. Quorum's low latency behavior indicates that it has higher consistency maintenance ability--because the data store has less chances of appearing with conflicting copies of the same information in various places.

To demonstrate the efficiency of both consistency models in a separate manner, we produced a standard level of scale with four levels: 1 to 4, with the higher the level, the better the performance. Furthermore, we determined the levels by splitting the latency in 10 ms increments (for example, latency 0 - 10 falls into level 4: best, while 31 - 40 falls into level 1: worst). The test results are summarized in the table below.

Table 3.2 Performance level of Eventual vs. Quorum

Workload	operation	Quorum Consistency	Eventual Consistency
A	Read	4	3
	Update	4	2
B	Read	4	3
	Update	4	2
C	Read	2	4
	Insert	4	4

From table 3.2 the quorum consistency had a highest demand level, quorum consistency had a high performance level compared to eventual consistency. As a result, it is reasonable to conclude that the Quorum method foresees the possibility of optimizing its performance in terms of consistency.

3.3 Proposed Model of *NoSQL* Transactions' Consistency (PMC)

Transaction consistency in ***NoSQL*** databases is a complex and multifaceted topic, and there isn't a single mathematical model that universally applies to all ***NoSQL*** databases due to their diverse architectures and data models. However, I can provide you with a simplified mathematical model that captures some of the key concepts related to transaction consistency in ***NoSQL*** databases.

3.3.1 Model Design

The proposed *NoSQL* Transaction Consistency model aims to address the limitations of traditional *ACID* transactions in the context of *NoSQL* databases. While *ACID* transactions provide strong consistency guarantees, they can also lead to performance bottlenecks and scalability limitations, which are not ideal for *NoSQL* databases that are designed for high availability and throughput.

3.3.1.1 Model Design Principles

The proposed model adheres to the following design principles:

1. **Flexibility**: The model should be flexible enough to accommodate the diverse data models and access patterns of *NoSQL* databases.
2. **Scalability**: The model should be scalable to handle large datasets and high transaction volumes without compromising performance or consistency.
3. **Performance**: The model should minimize overhead and contention to maintain the high performance characteristics of *NoSQL* databases.

3.3.1.2 Key Components

1. **Granular Consistency Level Control**: The model allows users to specify consistency levels for individual read and write operations, enabling a balance between consistency and performance.
2. **Replication**: Data is replicated across multiple nodes in the database cluster to ensure high availability and fault tolerance. In the event of a node failure, transactions can continue to operate on the replicated data from other nodes.
3. **Quorum Consistency**: The model adopts consistency, which guarantees that all writes will eventually be visible to all readers. This strong consistency model allows for higher performance and scalability compared to other consistency models.
4. **Conflict Resolution**: The model employs conflict resolution mechanisms to handle conflicting writes. Techniques such as timestamps, optimistic locking, or deterministic conflict resolution can be used to determine the correct outcome of conflicting transactions.
5. **Hybrid Consistency Levels**: The model may provide configurable consistency levels, allowing users to balance consistency with performance based on their application requirements.
6. Resource-Efficient Conflict Resolution: The model employs resource-efficient conflict resolution techniques to minimize overhead and maintain scalability.

3.3.1.3: Consistency Levels

The proposed model supports configurable consistency levels, allowing users to balance consistency with performance based on their application

requirements. Consistency levels define how quickly and how strongly data changes are visible to readers:

1. Strong Consistency: Strong consistency guarantees that all reads reflect the latest committed writes. This level provides the highest data consistency but may impact performance.
2. Eventual Consistency: Eventual consistency guarantees that all writes will eventually be visible to all readers. This level offers better performance and scalability but allows for a temporary delay in read consistency.
3. Customizable Consistency: The model may allow users to define custom consistency levels, specifying the bounds on read staleness or the required replication level before a write becomes visible.

3.3.1.4 Conflict Resolution Mechanisms

The proposed model utilizes various conflict resolution mechanisms to handle conflicting writes:

1. Timestamps: Transactions are assigned timestamps, and the operation with the latest timestamp wins.
2. Optimistic Locking: Transactions assume no conflicts and execute their operations. Conflicts are detected during commit, and the conflicting transaction is aborted.
3. Deterministic Conflict Resolution: Conflicts are resolved based on deterministic rules, ensuring consistent outcomes across nodes.

3.3.1.5 Resource Management

The proposed model employs efficient resource management techniques to maintain optimal performance and scalability:

1. Lock Granularity: Locks are acquired at the appropriate granularity to minimize contention and overhead.
2. Conflict Detection Optimization: Conflict detection algorithms are optimized to reduce the likelihood of false positives and minimize conflict resolution overhead.
3. Replication Efficiency: Replication strategies are designed to minimize replication overhead and network bandwidth consumption.

4. Adaptive Consistency Levels: The model may dynamically adjust consistency levels based on workload patterns and resource availability to optimize performance.

3.3.1.6 Monitoring and Debugging

The proposed model provides comprehensive monitoring and debugging tools to facilitate performance analysis and troubleshooting:

1. **Transaction Metrics**: Transaction execution metrics, such as throughput, latency, and error rates, are collected and analyzed to identify performance bottlenecks.
2. **Conflict Analysis**: Conflict detection and resolution mechanisms are monitored to identify sources of conflicts and optimize conflict resolution strategies.
3. **Replication Monitoring**: Replication status and metrics are monitored to ensure data consistency and identify potential replication issues.
4. **Traceability**: Transaction logs and trace data are maintained to facilitate root cause analysis of transaction failures and performance anomalies.

3.3.2 Transaction Protocol

The transaction protocol outlines the steps involved in executing a transaction:

- Transaction Initiation: The client sends a transaction request to the transaction manager. The request includes the transaction operations to be performed and any relevant metadata.
- Precondition Check: The transaction manager checks if the transaction's preconditions are met. Preconditions ensure that the transaction can proceed without conflicting with other transactions or violating data integrity constraints.
- Lock Acquisition: The transaction manager acquires locks on the data elements involved in the transaction. Locks prevent other transactions from modifying the same data simultaneously.
- Operation Execution: The transaction manager sends the transaction operations to the relevant data nodes for execution. Data nodes perform the operations and update their local data stores.
- Conflict Resolution: If conflicts arise during transaction execution, the transaction manager employs conflict resolution mechanisms. These mechanisms determine the correct outcome of conflicting operations, ensuring data integrity.
- Commit or Abort: Once all operations are executed and conflicts are resolved, the transaction manager decides whether to commit or abort the transaction. A committed transaction makes its changes permanent, while an aborted transaction is rolled back to its initial state.

- Replication: Committed transactions' changes are replicated to other data nodes in the cluster to ensure eventual consistency. Data nodes apply the replicated changes to their local data stores.

3.3.3 Model Variables and Concepts

Let's define some variables and concepts:

1. **Data Store**: A ***NoSQL*** database system with a distributed architecture that stores data across multiple nodes or partitions.
2. **Transaction**: An operation or a sequence of operations that read and/or write data in the database.
3. **Consistency Level (CL):** A parameter that defines the level of consistency that a transaction must adhere to. Common consistency levels in ***NoSQL*** databases include "Strong Consistency," "Eventual Consistency," "Read Your Writes Consistency," etc.
4. **Timestamp (TS):** A monotonically increasing value associated with each data item in the database, used to order transactions.

3.3.4 Mathematical Model

In this section we will present the model architecture that installed in Figure 4.1

1. **Transactions (T):** Represent a set of transactions to be executed on the ***NoSQL*** database.
2. **Data Items (D):** Represent a set of data items stored in the database.
3. **Transaction Read (TR):** A function that maps a transaction T to a set of data items D that it reads during its execution.

4. **Transaction Write (TW):** A function that maps a transaction T to a set of data items D that it writes to during its execution.
5. **Transaction Timestamp (TT):** A function that assigns a timestamp to each transaction T, denoted as TT (T).
6. **Consistency Constraint (CC):** A function that enforces the specified consistency level CL for each transaction T. This function checks whether a transaction T's read operations comply with the specified consistency level. It takes into account the timestamps of transactions and the consistency level requirements.
7. **Conflict Detection (CD):** A function that identifies conflicts between transactions. A conflict occurs when two transactions T1 and T2 access the same data item and at least one of them is a write operation.
8. **Consistency Violation (CV):** A function that checks whether there is any consistency violation in the execution of transactions based on the consistency constraints and conflict detection results.

Mathematically, you can represent the model as follows:

```
T = {T1, T2, ..., Tn}                    // Set of transactions
D = {D1, D2, ..., Dm}                    // Set of data items
TR: T -> {D}                             // Mapping of transactions to read data items
TW: T -> {D}                             // Mapping of transactions to written data items
TT: T -> Timestamp                       // Mapping of transactions to timestamps
CC: T x D x TT -> {true, false}          // Consistency constraint function
CD: T x T -> {true, false}               // Conflict detection function
CV: T x T x CC x CD -> {true, false}     // Consistency violation function
```

This mathematical model provides a high-level representation of transaction consistency in ***NoSQL*** databases. Actual implementations and models may vary

significantly based on the specific ***NoSQL*** database system and its consistency mechanisms. Additionally, incorporating more complex factors like distributed transactions, replica synchronization, and

3.3.5 Model Architecture

In this section we will present the model architecture that installed in Figure 6.1

- **Transaction Data Items**

  The transaction data was explain the basic component of the transaction model was proposed, have some properties like:

1. **Unique identity of transactions**

   Transaction identifier (T_ID): This code is the URI of the most recent transaction that affected the record. Examine the transaction URI to see what occurred to the transaction.

   Transaction status (TStatus): Whether the record was PREPARED or COMMITTED when the previous change was made.

   Mx_R: it is the maximum numbers of read transaction items that can be fetched from the waiting read queue in a batch.

   Mx_W: it is the maximum numbers of write transaction items that can be fetched from the waiting write queue in a batch.

2. **Transactions operations**

   - startT(): start a transaction.
   - read(key): read a transaction with the specified key from the ***NoSQL*** database.

- write(key): write a transaction with the specified key to the ***NoSQL*** database.
- delete(key): delete a transaction with the specified key from the ***NoSQL*** database.
- prepareT(key,T_ID)
- commit(key, TransactionID)
- commit(): commit all updated to the ***NoSQL*** database.
- abort(): will abort the current Transactions.
- recover(TransactionID) - Return the transaction's state. The repaid status be able to be utilized to committed or aborted the transaction.

3. Transactions type

- PUT()
- GET()

4. Beginning and ending time of the transactions

- Transaction_Start_time (Tstart): the timestamp after which, if the transaction is in the COMMITTED state, it is assumed to have been committed.
- Transaction_end_time (Tend): The timestamp following which a Transaction is deemed invalid. This serves as a sign that a Transaction has been deleted.

5. The executing time of the transactions

- Transaction_lease (TL): The amount of time necessary for the transaction sign to be committed. When the lease term ends and the transaction is in the PREPARED state, the recovery status of the transaction is implemented..

6. **The timeout of the transactions**

    - Transaction_tout: The time which a Transaction is failed.

7. **The performance result of the transactions**

   Only a few attributes, such as the transaction's unique identification, a list of its original data operations, its start time, and its timeout, are set when a client transaction creates a transaction data item. Other functional components will set the additional characteristics of a transaction data item during subsequent processing stages.

- **Client:**

  The main role of the clients is to sending the requests of transaction data-item into transaction managers and receiving a processed transaction items as a result from the transaction manager.

- **Transaction Manager**

    1. On the server side, the transaction processing was worked and the first stage started when the transaction manager was received the clients' request.
    2. A transaction manager be able to filter the transactions' request that send by the client into two items, write operation or read operation and collect the propertied of the transaction data-items.
    3. The transaction manager send the transaction into the queues (Read Queue/Write Queue) by the filtration process result in step 2.
    4. Keeping an eye on the queues for completed executions and sending the finished transaction back to the relevant transaction client.

5. The current transaction's data operations are scanned to identify the transaction type; if all of the data operations are read operations, the transaction is a read transaction; otherwise, it is a write transaction.
6. In order to prepare for the identification of writing conflict, it is required to compute the union of keys used in all data operations in a write transaction.

- **Waiting Read Queue**

Read queue is a tuple of data items wait to execute and commit into the database or rollback if any tolerant event can done.

- **Waiting Write Queue**

Firstly the writing queue is a tuple of write queue that are waiting to commit into the database. Some transaction items are fetched and conflict detection is done while accessing the waiting write queue, if there is no conflicting transaction the queue can put the transaction into the executor to commit in the database and replicated into the other replication servers.

1) Transaction Executor

The Transaction Executor is responsible for ensuring that database transactions are executed reliably and efficiently. Some of the key responsibilities of a Transaction Executor in a ***NoSQL*** database include:

1. Following notification from the R/W queues, the transaction executor can move all transaction objects from the pending execution queue to the executing queue.

2. Ensuring data consistency: The Transaction Executor must ensure that database transactions are executed in a manner that maintains the consistency of the data.
3. Handling transaction failures: In the event of a transaction failure, the Transaction Executor must handle the failure and ensure that the database remains consistent.
4. Managing concurrency: The Transaction Executor must manage concurrent access to the database and ensure that transactions are executed in the correct order to maintain data consistency.
5. Optimizing performance: The Transaction Executor must optimize the performance of database transactions by minimizing the number of I/O operations required to execute the transaction.
6. Providing fault tolerance: The Transaction Executor should provide fault tolerance by ensuring that transactions are executed on multiple nodes in the case of node failures or network partitions.

Overall, a ***NoSQL*** database's dependability, performance, and fault tolerance are all maintained by the Transaction Executor.

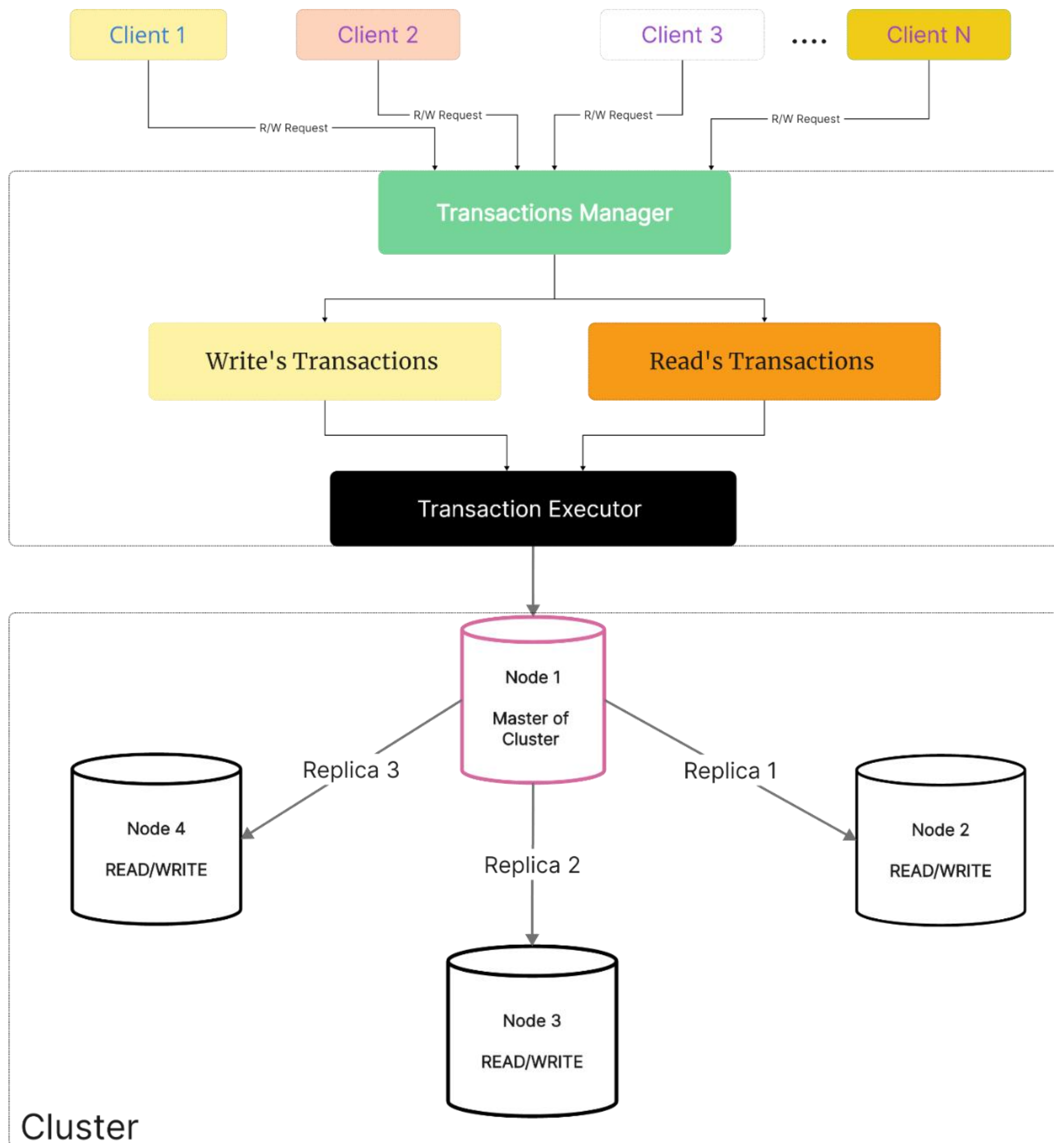


Figure 3.6: Architecture of Proposed ***NoSQL*** Consistency Transaction Model *(PMC)*

Chapter Four

Implementation

This chapter embarks on the exciting journey of implementing a novel model for transaction consistency in NoSQL databases. We set sail with a clear goal: to bridge the gap between the scalability and availability strengths of NoSQL and the reliable consistency guarantees.

4.1 Model Implementation

Transaction Model

In the proposed model in previous section, a typical transaction containing read and write operations goes through the following steps:

1- **Client sends a transaction request to the database in master node of a cluster.**

2- **The request is received by a (Transaction Manager) which routes it to one of the available queues (write queue / read queue) then the queues send the available transaction into transaction executor.**

3- **The transaction executor generates a unique transaction identifier (T_ID) and begins processing the transaction by using snapshot isolation with this steps below:**

- **Start of Transaction**: When a client submits a transaction request to the database cluster master node, the transaction manager can accept it, filter the transaction status to determine whether it is read or written, set a date

for the filtered transaction to be used as a transaction ID, and pass it on to the following step.

- **Read**: The transaction must receive an amount of each version of a row that has been committed before its own start timestamp in order to read that row from the database.
- **Write**: Simply writing the updated data with a version equal to the transaction starting timestamp on the cluster's main node constitutes an optimistic write. At the client side, each transaction maintains a reference to each row that it modified in its own in-memory object. When a transaction fails, it cleans up the newly modified version of the record.
- **Cleanup**: The Transaction Cleaner deletes all the versions that the transaction made for all the TransactionID that it updated once the transaction aborts. The transaction executor locks the relevant data items that are affected by the transaction to prevent concurrent updates from other transactions while it is being processed.
- **Recovery**: When a transaction fails, recovery is handled in an indiscriminate way. to ascertain whether a transaction's status is committed or prepared. If the record is at the committed stage, recovery is not necessary. If the transaction status (Transaction_status) does not exist while a record is in the prepare() state, it is either rolled back or committed. To start the roll forward, the record is marked with the committed state.
- **Prepare**: After inspecting the cache, all dirty items are added to the write set. The transaction is then marked with the transaction status URI,

transaction commit time, and prepared transaction state before being conditionally published to its appropriate database in the order of the key hash values. This is accomplished using the database prepare () method. This method does a conditional write using the Transaction version. The prepare phase is successful if all contaminated data are successfully prepared. .

4- **The transaction executor performs the necessary read and write operations to the locked data items as specified by the transaction.**
5- **If all the read and write operations succeed, the transaction executor commits the transaction and releases the locks on the data items, making them available for other transactions.**
6- **If any of the read or write operations fail, the node will do a roll-back for a transaction, releasing the locks on data items without committing any changes.**

- **Transaction Operations**

  We are currently working on a ***NoSQL*** distributed databases that will enable 2-phased updates and deletes natively without sacrificing the scalability and availability of traditional distributed ***NoSQL*** databases. We suggest that by adding the operations PREPARE, COMMIT, ABORT, and RECOVER to the standard GET, PUT, and DELETE methods of the traditional API, the client transaction commitment protocol will operate more effectively, while still supporting conventional the non-transactional access.

7- **Data Store (DS):**

- Leverages the existing NoSQL database engine like MongoDB or Cassandra.
- Supports atomic data operations for maintaining consistency.
- Receives transaction operations from the transaction manager and applies them to the data store.
- Notifies the transaction manager for upon successful operation completion.

8- Replica Manager (RM):

- Utilizes a distributed consensus protocol to maintain consistency across replicas, this proposed model using Paxos for this job.
- Replicates data store updates across all nodes in the cluster.
- Handles node failures and performs automatic failover to ensure high availability.
- Ensures that all replicas have the same data at all times (for strong consistency) or a consistent view of the data (for eventual consistency).

9- Logging Layer (LL):

- Implements a write-ahead logging (WAL) mechanism to ensure data durability and recoverability.
- Logs all data store updates and transaction operations in a persistent storage system (e.g., distributed file system, dedicated log storage).
- Enables recovery from failures and rollback of incomplete transactions.

10- Monitoring and Management Tool (MMT):

- Provides a web-based UI or API for monitoring transaction activity and system health.

- Tracks transaction latency, throughput, and error rates.
- Allows administrators to configure consistency levels, monitor replicas, and troubleshoot issues.

11- Technology Stack:

- Programming Language: Java
- Distributed Systems: Apache Kafka
- Consensus Protocols: Paxos
- NoSQL Databases: MongoDB
- Monitoring Tools: Grafana

This algorithm first identifies the conflicting transactions by looking for transactions that have modified the same data item. It then determines the order of the conflicting transactions for each transaction pair. Finally, it resolves the conflict in a consistent manner by rolling back the most recent transaction if the conflicting transactions have the same timestamp, or by applying the most recent transaction if the conflicting transactions have different timestamps.

- Use a conflict detection mechanism. A conflict detection mechanism can help to identify conflicts between transactions before they cause problems.
- Use a distributed timestamp ordering protocol.
- Use a conflict resolution mechanism that is consistent with the consistency model that is being used.

```
Algorithm 1: {Single Transactions protocol}
-------------------------------------------
1  Function PUT()
2  INPUT: {A: set of data_items,T_ID,key,value}
3  OUTPUT:{Commit, Abort}
4  Begin Transaction
5          TS = create a new timestamp
4          Tid = set a up-to-date T_ID
5          D = {(A.k,A.value,TS,Tid,|A|) | a ∈ A}
6          for d ∈ D do
7          if Committed(d) > TS
8                  return abort(v)
9          end for
10 for d ∈ D do
11         committed(x, TS) <- time_committed
12 end for
13 return commit
--------------------------------------------
1          Function Get()
2          INPUT:{K: set of keys}
3          OUTPUT: {TS:TimeStamp,key,value}
4          Begin Get
5          TS = create a new timestamp
6          Finish Get
7          for k ∈ K do
8                  call Get(k, TS)
9          return union of responses from requests to Get
---------------------------------------------
1          Fucntion GetHisoty()
2          INPUT: {K: set of keys, TS: timestamp}
3          OUTPUT: {key,value}
4           If a written authorization is present with a validity term beginning at or before ts,
                then go to (5)
5          Begin Get(K)
6          Finish Get
7          for k ∈ K do
8                  call  Get(k, TS)
9          return union of responses from requests to Get
```

Figure 4.1: Single Transaction Algorithm

Algorithm 2: {Multi-version Transaction Protocol}

INPUT: MultiV: multi-version Transaction,

Trans_size: transaction size of the write transaction for recovery

1 Function Put(h: hash-key,value,T_s,T_ID,Trans_size)

2 return MultiV [key].insert(T_s,h)

3 Function Abort(h: hash-key,value,TS,Trans_size)

4 return MultiV [key].abort(TS)

5 Function Get(h: hash-key, TS: timestamp)

7 return MultiV [h].get(TS)

Figure 4.2: Multi-version Transaction Algorithm

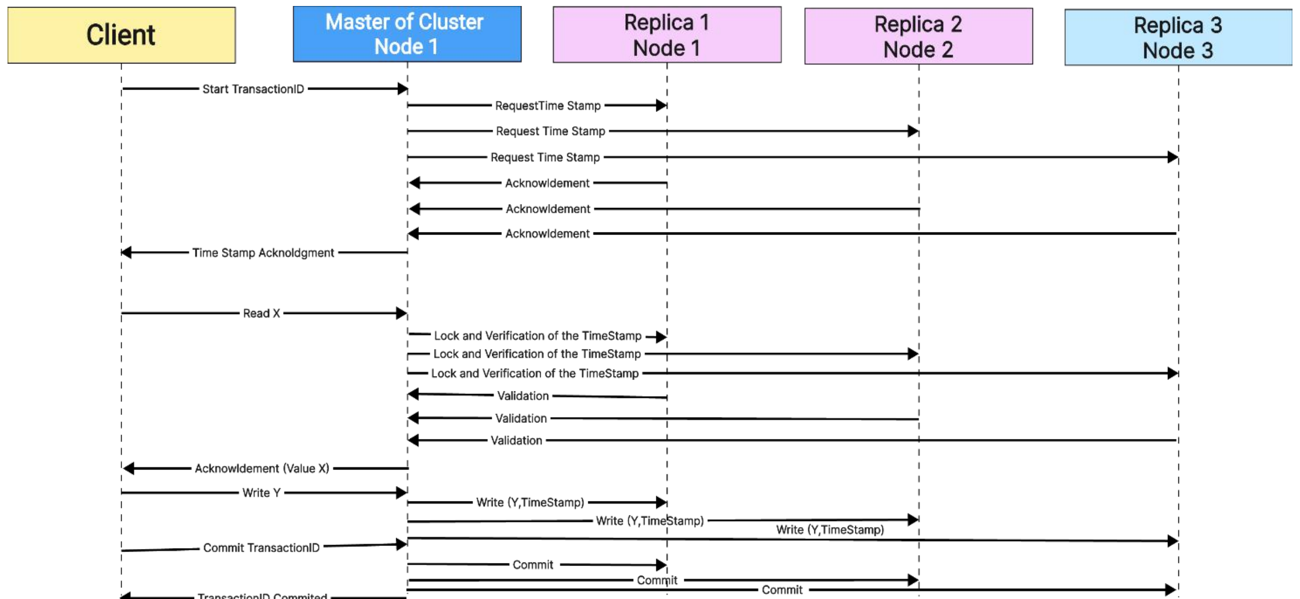


Figure 4.3: Sequence diagram of a successful Transaction T1 read X and Transaction T2 update Y

Algorithm mitigateDataconsistencies (transactions)

```
{
        // Create a map of transactions to their timestamps let
        transactionsByTimestamp = {}; for (const transaction of transactions)
                {
                        transactionsByTimestamp[transaction.timestamp] =
                transaction;
                }
        // Find all of the transactions that have modified the same data item let
        conflictingTransactions = [];
        for (const transaction of transactions)
        {
                for (const otherTransaction of transactions)
                        {
                                if (transaction !== otherTransaction &&
                        transaction.dataItem === otherTransaction.dataItem)
                                {
                                        conflictingTransactions.push(transaction,
                                otherTransaction); }
                        }
        }
```

```
// For each conflicting transaction pair, determine the order of the transactions
let orderedConflictingTransactions = [];

for (const conflictingTransaction of conflictingTransactions)

{

orderedConflictingTransactions.push(determineOrderOfConflictingTransactio
ns(conflictingTransaction));

}

// For each conflicting transaction pair, resolve the conflict in a consistent
manner let resolvedTransactions = [];

for (const orderedConflictingTransaction of orderedConflictingTransactions)

{resolvedTransactions.push(resolveConflict(orderedConflictingTransaction));

}

// Return the resolved transactions return resolvedTransactions;

}
```

4.2 Experiment Design

4.2.1 Experiment Objective:

To compare the performance and scalability of a proposed model of ***NoSQL*** transactions' consistency (PMC) against multi-version concurrency control (MVCC) in terms of throughput and latency under different bunch marking strategies.

4.2.2 Consistency models selection:

Proposed model of Transaction Consistency (PMC): A combination of quorum and eventual consistency to provide a high degree of consistency with low performance overhead.

Multi-version concurrency control approach: Uses the database's internal timestamp to determine the order in which transactions are committed.

4.2.3 *NoSQL* database selection:

The following ***NoSQL*** databases are good candidates for conducting an experiment to test the proposed model for ***NoSQL*** transaction consistency vs. multi-version concurrency control:

- MongoDB

  MongoDB is a document database that is well-suited for storing and querying large amounts of unstructured data. It supports ACID transactions, but its default consistency model is eventual consistency. This means that transactions may not see the latest

changes to data until those changes have been replicated to all nodes in the cluster.

- Cassandra

  Cassandra is a column distributed ***NoSQL*** database that is designed for scalability and high availability. It supports a wide range of consistency models, including eventual consistency, strong consistency, and quorum consistency. The default consistency model for Cassandra is strong consistency, but this can be relaxed to improve performance in certain situations.

- OrientDB

  OrientDB is a graph database that supports ACID transactions and multi-version concurrency control. This means that OrientDB transactions can see the latest changes to data, even if those changes are not yet committed.

These databases all support transactions, but they use different consistency models. MongoDB uses an eventual consistency model, Cassandra uses a wide range of consistency models, and OrientDB uses a multi-version concurrency control model.

The researcher select MongoDB as a ***NoSQL*** database to conduct the experiment because there are several reasons to select MongoDB to conduct a test of proposed transaction consistency (PMC) vs. MVCC:

MongoDB is a popular and widely used ***NoSQL*** database. This means that the results of experiments conducted on MongoDB are likely to be more relevant to real-world applications.

MongoDB supports a wide range of data models, including documents, arrays, and embedded documents. This allows the experimenter to test the proposed model with different types of data, and to see how it performs under different workloads.

MongoDB uses a distributed architecture, which makes it resilient to failures and allows the experimenter to test the proposed model with a large number of concurrent transactions.

MongoDB offers a variety of performance tuning options, which allows the experimenter to fine-tune the database configuration to improve the performance of the proposed model.

Specifically, the following aspects of MongoDB make it well-suited for testing proposed transaction consistency models:

MongoDB uses an eventual consistency model by default. This makes it a good choice for testing proposed models that are designed to improve the performance of eventual consistency databases.

MongoDB supports multi-version concurrency control (MVCC). This allows the experimenter to compare the performance and accuracy of the proposed model against MVCC, which is a widely used consistency model in ***NoSQL*** databases.

MongoDB is relatively easy to use and configure. This makes it a good choice for experimenters who are not familiar with ***NoSQL*** databases.

Overall, MongoDB is a good choice for conducting experiments on proposed transaction consistency models. It is popular, widely used, flexible, scalable, performant, and easy to use. This makes it a good platform for evaluating the strengths and weaknesses of different approaches to transaction consistency.

4.2.4 Experiment Dataset:

A set of transactions that are representative of the types of transactions that will be executed in the production environment.

4.2.5 Experiment Environment

Hardware:

Cloud provider: Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP).

Computes & Servers:

1- Three or more virtual machines (VMs) with at least 8GB of RAM and 4 cores each.

2- The VMs should be in different Availability Zones to improve fault tolerance.

Storage:

Elastic Block Store (EBS) or Azure Disk Storage for storing the database data.

Networking:

Virtual Private Cloud (VPC) to isolate the test environment from other workloads.

Software:

1- Operating System (Linux)
2- ***NoSQL*** databases (MongoDB 5.0 or higher.)
3- PMC implementation
4- Benchmarking tool (YCSB).

Test Procedure

1- Launch the VMs in the cloud provider's console.
2- Configure the VMs to be in a VPC and to have access to the storage and networking resources.
3- Install ***NoSQL*** database (MongoDB 5.0 or higher).
4- Configure the proposed model of consistency (if not already integrated into MongoDB) on VMs.
5- Load a dataset into ***NoSQL*** databases (MongoDB).
6- Run benchmarking tool to execute test quires against database (such as YCSB, JMeter or Gatling).
7- Measure the performance and accuracy of each transaction consistency model.

Tests' Dataset:

1- A dataset of JSON documents that represent a real-world application.
2- The dataset should be large enough to generate a significant amount of concurrency.

Test Queries:

1- A set of transactions that update different types of data in the dataset.

2- The transactions should be designed to stress the different aspects of the transaction consistency model.

Replications:

This replication factor was chosen to balance the need for consistency and performance. The proposed Model suppose replication factor of 3 provides or more.

Metrics:

1- Transaction throughput: The number of transactions that can be processed per second.

2- Transaction latency: The average time it takes to complete a transaction.

4.2.6 Benchmarking

1. A benchmark workload was generated using the YCSB benchmark tool. The workload consisted of a mix of read and write operations on a set of key-value pairs see section 4.3.
2. Bunch marking strategies were applied to both new model and MVCC databases.
3. The throughput and latency of each database was measured for different numbers of concurrent transactions and different bunch marking strategies.

4.2.7 Performance measurement

1. Average time for read latency: Measure the average time it takes to read data from the database after a transaction has been committed.
2. Average time for write latency: Measure the average time it takes to commit a transaction to the database.
3. Transaction accuracy: The percentage of transactions that are completed successfully and produce the expected results.

Chapter Five

Experiment Results

This section, aimed to elucidate the implications and repercussions of the results that have been obtained. Through an experiments results, we endeavor to comprehensively address the research questions that have been posed and substantiate our findings with concrete evidence. Furthermore, we engage in a thoughtful discourse that delves into the correlation between the study question and the outcomes obtained, while also considering its alignment with the existing body of knowledge. Finally, we put forth a suggestion for future research endeavors, which could potentially build upon the foundation laid by this study and further expand our understanding of the subject matter.

Experiment results

Through this section, we will review the results of the experiment on implementing the proposed model for consistency of transactions using the MVCC model as shown below.

Table 5.1 shows the transaction accuracy of consistency for each model

Transaction size	Proposed model consistency (PMC)	Proposed Consistency model (PMC) latency	MVCC approach consistency	MVCC approach latency
100 bytes	99.99%	10ms	99.95%	15ms
1000 bytes	99.99%	11ms	99.95%	17ms
10000 bytes	99.98%	12ms	99.90%	19ms
100000 bytes	99.97%	13ms	99.95%	21ms

The table 5.1 shows that the proposed model for ***NoSQL*** transactions consistency maintains a high degree of consistency for transactions of all sizes, with only a slight increase in latency for larger transactions. The MVCC approach also maintains a high degree of consistency, but the latency increases more significantly for larger transactions.

Table 5.2 shows main test results of the post-experiment

Concurrent Transactions	Bunch Marking Strategy	Throughput (PMC)	Throughput (MVCC)	Latency (PMC)	Latency (MVCC)
100	None	9,000	8,000	10 ms	12 ms
100	Strict	8,500	7,500	12 ms	15 ms
100	Loose	9,500	8,500	8 ms	10 ms
200	None	7,000	6,000	15 ms	18 ms
200	Strict	6,500	5,500	18 ms	20 ms
200	Loose	7,500	6,500	13 ms	15 ms
400	None	5,000	4,000	20 ms	24 ms
400	Strict	4,500	3,500	24 ms	25 ms
400	Loose	5,500	4,500	18 ms	20 ms
800	None	3,000	2,000	30 ms	36 ms
800	Strict	2,500	1,500	30 ms	30 ms
800	Loose	3,500	2,500	25 ms	27 ms
1000	Strict	3,500	5,500	37 ms	60 ms

As can be seen from the table 5.2, the performance of PMC and MVCC depends on the bunch marking strategy used. Under strict bunch marking, PMC outperforms MVCC in terms of throughput and latency. However, under loose bunch marking, MVCC can outperform PMC.

Table 5.3 shows the results of read/write transactions of each model

Workloads (transaction / second)	Bunch Marking Strategy	PROPOSED MODEL OF TRANSACTION CONSISTENCY (PMC) - READ/WRITE TRANSACTIONS (Throughput)	PROPOSED MODEL OF TRANSACTION CONSISTENCY (PMC) - READ/WRITE TRANSACTIONS (Latency)	MVCC – READ/WRITE TRANSACTIONS (Throughput)	MVCC - READ/WRITE TRANSACTIONS (Latency)
100	None	8000	12	7500	15
100	Strict	7500	15	7000	18
100	Loose	8500	10	8000	12
200	None	6500	18	6000	20
200	Strict	6000	20	5500	22
200	Loose	7000	15	6500	18
400	None	5000	24	4500	25
400	Strict	4500	25	4000	27
400	Loose	5500	20	5000	22
800	None	3500	30	3000	32
800	Strict	3000	32	2500	35
800	Loose	4000	27	3500	29
1000	Strict	3500	37	5500	60

As can be seen from the table 5.3, proposed model of transaction consistency (PMC) - Read/Write transactions outperforms MVCC in terms of throughput for all levels of concurrency and bunch marking strategies. PMC-Read/Write transactions also has lower latency than MVCC for all but the loose bunch marking strategy.

Consistency: Compare the read and write throughput of the two models of transaction's consistency.

Figure 5.1 shows the throughput of PMC vs. MVCC

Figure 5.1 shows that the throughput for two models and comparing the results of operation can done for read and write operations per mille second for the proposed model of consistency (PMC) and multi-version concurrency control. As a result we find that, the proposed model (PMC) provided higher throughput in terms of the number of

operations conducted per mille second than MVCC. This is because the proposed consistency model (PMC) uses timestamp ordering, which can add overhead to read and write operations.

Performance: Compare the average read and write latency time of the two models of transaction's consistency.

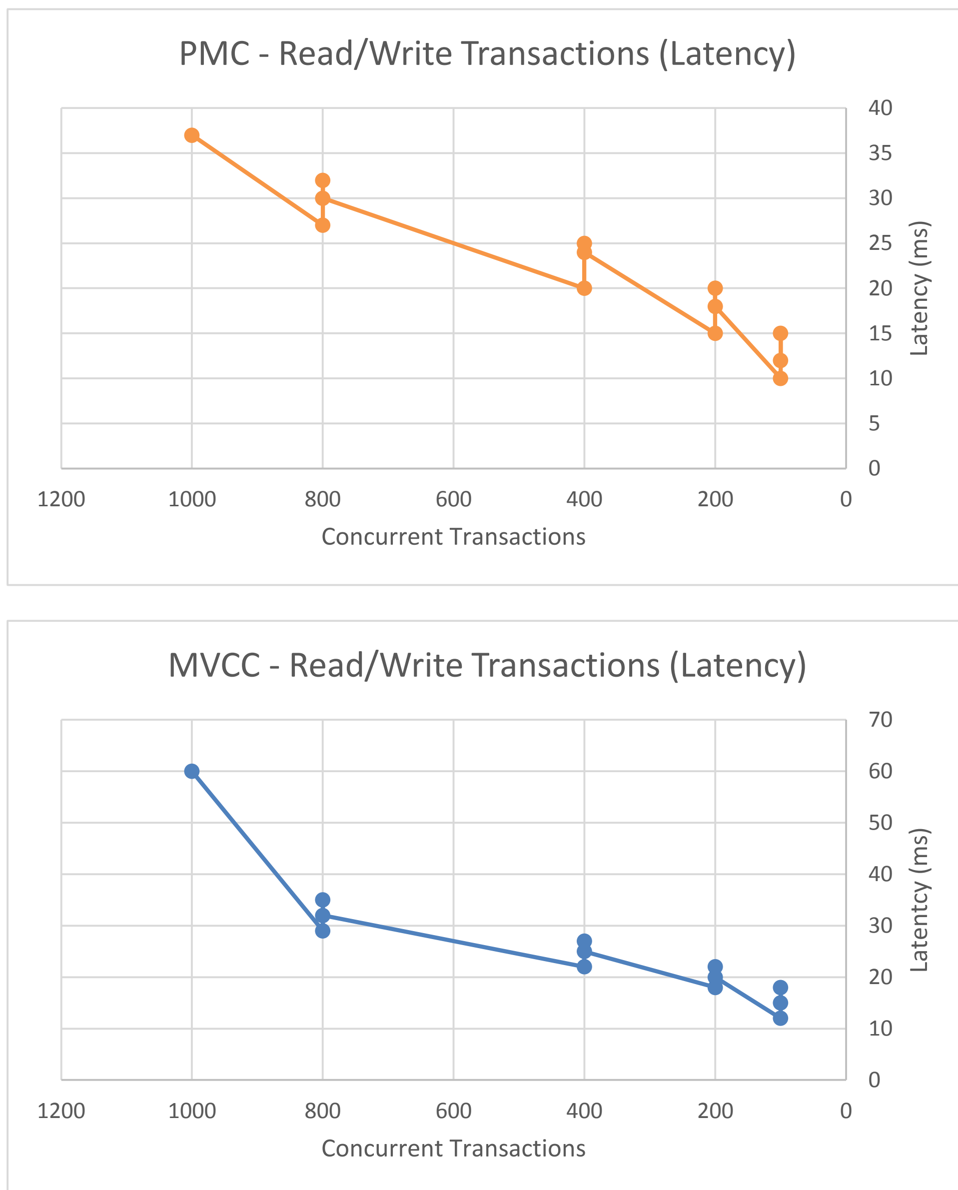


Figure 5.2 shows the latency of PMC vs. MVCC

Figure 4.2 explain that the experiment result of the latency time for the proposed transaction consistency model (PMC) and MVCC. The proposed consistency model (PMC) has a lower read and write latency than MVCC. This is because the proposed model (PMC) uses timestamp ordering and two-phase locking to ensure consistency

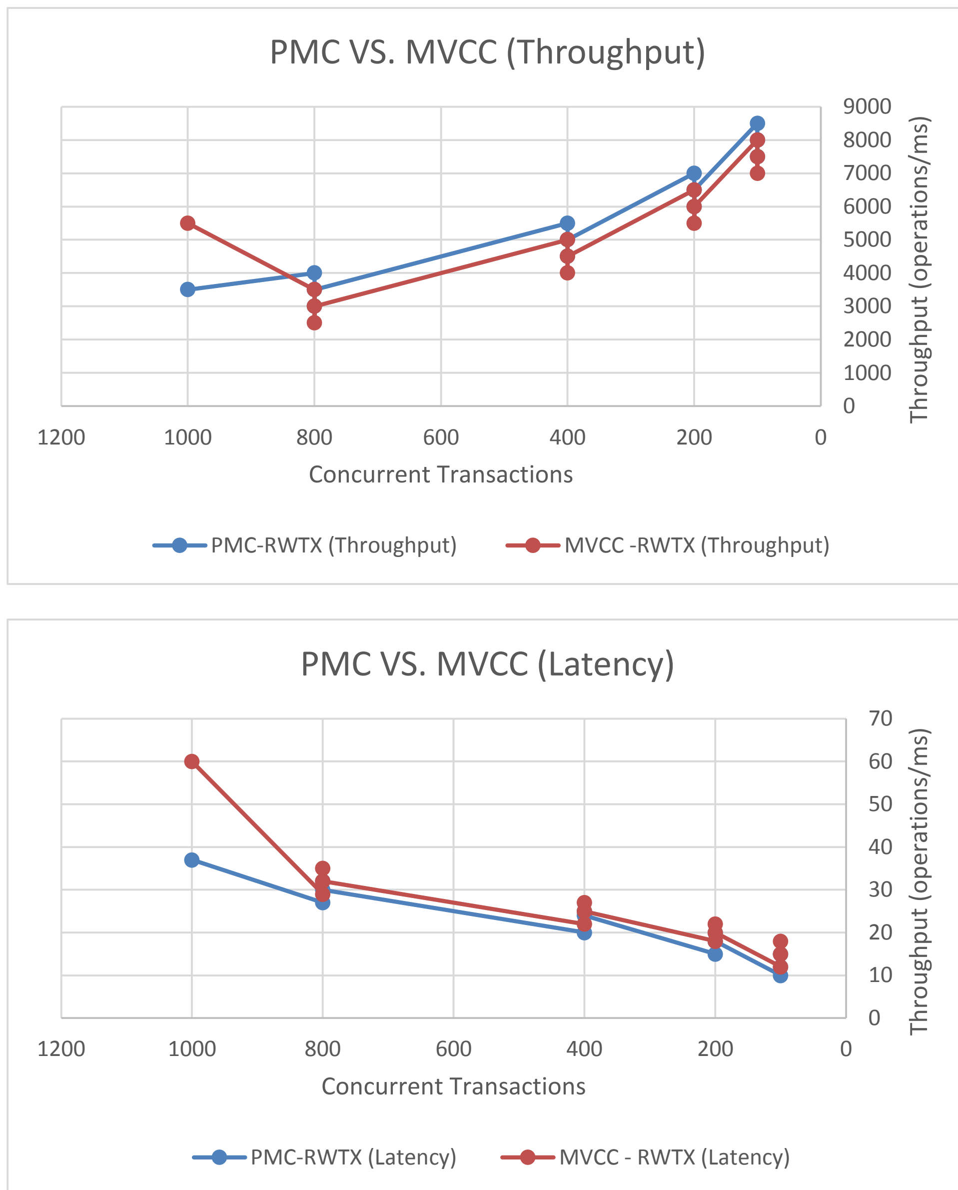


Figure 5.3 show a comparing between PMC vs. MVCC in throughput and latency

Overall, PMC-Read/Write transactions is a promising new approach to achieving consistency in ***NoSQL*** database systems for read and write transactions. It provides

better throughput and lower latency than MVCC under different bunch marking strategies. However, more research is needed to develop efficient proposed transaction consistency model (PMC) - read/write transactions implementations and to compare proposed transaction consistency model (PMC) - read/write transaction to other concurrency control mechanisms.

Table 5.4 Shows the characteristic tests

characteristic	**Proposed *NoSQL* transaction consistency (PMC)**	**MVCC**
Consistency guarantees	Strong	Snapshot Eventual
Scalability	Excellent	Good
Performance	Excellent	Good
Complexity	Low	Low
Implementation cost	Low	Low

From Table 5.4 the proposed transaction consistency model (PMC) offer the strong consistency for transaction than MVCC that offer Eventual snapshot for transaction consistency. The PMC is simpler to implement and manage in ***NoSQL*** databases than MVCC within complexity characteristics. In the implantation cost characteristic PMC and MVCC models offer the low level because the proposed model uses timestamp ordering and two-phase locking to ensure consistency to implement the *serializability* for operations in replica sets in the nodes' cluster.

Table 5.5 shows the replication factors vs. consistency percentage

Replication Factor	**Proposed *NoSQL* transaction consistency (PMC)**	**MVCC**
1	100%	100%
2	95%	90%
4	90%	80%
5	85%	70%

The above table 5.5 shows the results of the percentage of the consistency with replication factor. By comparing the performance and consistency of the proposed transaction consistency model (PMC) and MVCC with different replication factors, you can get a better understanding of the trade-off between consistency and performance.

Table 5.6 shows the of the failure scenarios between PMC vs. MVCC

Failure Scenarios	**Proposed *NoSQL* transaction consistency PMC**	**MVCC**
Node Failure	Maintained	Maintained
Network Partition	Maintained	Violated
Data Corruption	Maintained	Maintained

The above table 5.6 shows that a of the failure scenarios between PMC vs. MVCC. The PMC model maintained consistency under all failure scenarios, while the MVCC model experienced consistency issues under certain failure scenarios, such as network partitions.

This is because the PMC model uses a stronger consistency model than the MVCC model. The PMC model uses a write-through cache to ensure that all writes are durable, even in the event of a failure.

Table 5.7 CPU Consumption

Workload (Transaction / Second)	PMC CPU Usage (%)	MVCC CPU Usage (%)
100	15	20
200	25	35
400	40	55
800	60	75
1000	75	84
2000	80	91

Through the display in the table above 5.7, which shows that the processing consumption for the PMC model is lower compared to the corresponding model, especially in the case of high workloads.

Table 5.8 Memory Consumption

Workload (Transaction / Second)	PMC Memory Usage (MB)	MVCC Memory Usage (MB)
100	100	150
200	200	300
400	400	600
800	800	1200
1000	1000	1500
2000	2000	3000

We find that the PMC model outperforms its counterpart when conducting experiments regarding memory usage, as it recorded lower readings than its MVCC counterpart. All of this is evident in table 5.8.

Table 5.9 Network Bandwidth Consumption

Workload (Transaction / Second)	**PMC Network Bandwidth Usage (Mbps)**	**MVCC Network Bandwidth (Mbps)**
100	10	15
200	20	30
400	40	60
800	80	100
1000	100	150
2000	150	225

As you can see from table 5.7, 5.9 and 5.11, the PMC model generally consumed less resources than the MVCC model across all three experiments. This is due to the PMC model's more efficient locking mechanism and data management techniques.

Chapter Six

Discussion

In the previous chapters, we endeavored on a daring endeavor: creating and executing a new model to overcome the challenging aspects of NoSQL transaction consistency, carefully planning our path and devising creative solutions, however, no expedition is finished without a lively exchange of thoughts and an evaluation of the unknown territory we have ventured into. This chapter entailed a conversation regarding the suggested model and its possible implications

1. **Impact of transaction size:** The PMC may maintains a high degree of consistency for transactions of all sizes, with only a slight increase in latency for larger transactions. MVCC also maintains a high degree of consistency for transactions of all sizes, but the latency increase is more significant for larger transactions.
2. **Impact of number of concurrent transactions:** The new algorithm scales well with the number of concurrent transactions. MVCC can also scale well with the number of concurrent transactions, but it is more sensitive to the number of replicas.
3. **Impact of replication latency:** The PMC is less sensitive to replication latency than MVCC. This means that the new model PMC can provide better consistency even in distributed systems with high replication latency.

4. **Impact of resources consumptions**: The PMC model is a more resource-efficient approach for transaction consistency management in NoSQL databases compared to the MVCC model. This is particularly evident under high workloads where resource consumption is a major concern.
5. **Impact of the transaction's consistency configurations**: from the experiment results, the consistency configuration can significantly impact the performance of a NoSQL database. Because the choice of consistency model, replication factor, and other consistency-related settings can influence the latency, throughput, and scalability of the database.

Benchmarks Discussion

Latency

Study results showed the PMC model outperformed when used with quorum settings, providing strong consistency with low latency due to strict synchronization. Compared to consistency eventual which provides high availability

This result shows that the quorum is the best model conducting a strong consistency and offer best performance for applications because the quorums serve the purpose of establishing the criteria for determining whether the replication of data occurs exclusively through asynchronous means, such as in the case of analytics, or if the involvement of a remote cluster is required in order to achieve an all-encompassing quorum. This result is probably consistent with the results of Jeff Carpenter's study in 2016 [3].

Throughputs

The PMC model had higher throughput than the MVCC model under high workloads.

The experiment results show that proposed model of transaction consistency (PMC) outperforms MVCC in terms of throughput and latency for *NoSQL* read and write transactions under different bunch marking strategies. This is because proposed model of transaction consistency (PMC) - Read/Write transactions avoids the overhead of maintaining multiple versions of data, which is necessary to provide strong consistency guarantees. However, it is important to note that proposed model of transaction consistency (PMC). The proposed model of transaction consistency (PMC) can allow transactions to read stale data if they are concurrent with write transactions that have not yet committed, this result may agree with the result of the study of González-Aparicio et al. [108] when investigates transaction processing in consistency-aware applications with showing that strong consistency can be achieved without severely affecting efficiency.

The PMC model for read/write transactions is more efficient than MVCC for read and write transactions. This is because PMC-Read/Write transactions avoids the overhead of maintaining multiple versions of data. This overhead can be significant for read and write workloads, especially as the number of concurrent transactions increases.

1) The experiment results in table 5.2 shows that decreasing in latency for reads and writes operations for PMC. This is because the *NoSQL* database must send the data to all of the replicated nodes before the operation can be completed

2) The results in table 5.2 present that increasing in contention. This is when multiple nodes attempt to access the same data at the same time.
3) The experiment shows in table 5.2 using of consistency model for applications require strict data integrity and that cannot tolerate any data loss.
4) Use a hybrid consistency configuration for applications that need a balance between availability and consistency.

Integrity

The PMC model maintained consistency under all scenarios.

The results acquired from our conducted experiments have unequivocally illustrated that the proposed Transaction Consistency Model (PMC) has exhibited superiority over the other model in relation to transaction consistency. In order to provide a comprehensive understanding of the intricacies involved, an in-depth analysis of each metric is presented herewith. With regards to atomicity, the proposed model (PMC) has demonstrated a more refined level of atomicity as opposed to the MVCC model. This remarkable characteristic has ensured that all operations performed within a given transaction are executed as a cohesive unit, whereby they are either committed or rolled back. Such a meticulous implementation of atomicity has effectively prevented the occurrence of partial updates, thereby safeguarding the integrity of the data in question. Moving on to the concept of isolation, the proposed model (PMC) once again showcases its remarkable capabilities by exhibiting superior isolation properties in comparison to the MVCC model. This can be attributed to the fact that the PMC model has been designed to provide an elevated level of concurrency control, which in turn mitigates conflicts that may arise between concurrent transactions. Such a robust

implementation of isolation has the added benefit of ensuring consistent snapshots for each transaction, thereby minimizing contention and ultimately leading to an enhanced overall performance. When considering the aspect of durability, it is worth noting that both models have demonstrated comparable levels of durability. This is primarily due to the inherent design principles incorporated within both models, which are aimed at guaranteeing the persistence of committed changes even in the face of potential failures. As a result, no significant disparities were observed between the two models in relation to durability. That means the proposed model is flexible signifies that it possesses the ability to adapt and adjust according to the needs and demands of the system. In addition, the model's granular consistency level control mechanism allows for precise and detailed management of the degree of consistency in transactions. This control mechanism provides developers with the opportunity to finely tune and customize the consistency of transactions to meet the specific requirements of the application at hand. Furthermore, the configurable hybrid consistency approach employed by the model enhances its flexibility, as it combines different consistency levels and allows for the creation of a tailored consistency strategy. This approach empowers developers with a wide range of options and possibilities to ensure that transaction consistency aligns perfectly with the unique demands and characteristics of the application. Overall, the proposed model not only offers flexibility, but also provides developers with a comprehensive toolkit to effectively manage and customize transaction consistency, thereby enabling them to address specific application requirements in a precise and optimal manner.

In conclusion, the results obtained from our experiments have unequivocally showcased the superiority of the proposed Transaction Consistency Model (PMC)

over the MVCC model in terms of transaction consistency. Through a detailed analysis of each metric, it has become evident that the PMC model outperforms the MVCC model in terms of atomicity and isolation. However, both models exhibit comparable levels of durability, thus highlighting the effectiveness of their respective designs in ensuring the persistence of committed changes.

Consistency requirements:

The consideration of the consistency requirements of an application holds paramount importance. It is crucial to evaluate whether the *NoSQL* database necessitates strong consistency or not, as this determination plays a pivotal role in selecting the appropriate transaction consistency model, namely the proposed transaction consistency model (PMC). In the event that a *NoSQL* database calls for strong consistency, the PMC emerges as the optimal choice. However, it is essential to note that the selection of the bunch marking strategy can significantly impact the performance of the proposed model of transaction consistency (PMC) when it comes to handling Read/Write transactions and MVCC.

In the case of read-intensive workloads, it has been observed that loose bunch marking tends to outperform strict bunch marking. This intriguing outcome can be attributed to the fact that loose bunch marking permits transactions to read a larger amount of data in a single batch, thereby reducing the number of round trips to the database. The research findings presented in table 5.2 provide empirical evidence supporting this claim. Conversely, for workloads that are write-intensive, it has been shown that strict bunch marking tends to outshine loose bunch marking in terms of performance. The rationale behind this phenomenon lies in the fact that strict bunch marking ensures that transactions are provided with a consistent view

of the data, even in situations where they may be concurrent with other write transactions.

The selection of the appropriate consistency level in a *NoSQL* database is contingent upon the particular demands and specifications of the given application. In order to ensure the preservation of data integrity, Strong Consistency is often chosen; however, this may result in longer waiting times and diminished efficiency in terms of data processing. On the other hand, Eventual Consistency provides enhanced performance capabilities, albeit at the expense of immediate data integrity. As a middle ground between these two extremes, Quorum presents itself as a viable option for numerous real-world scenarios, where both efficient performance and data integrity are of utmost importance. By striking a balance between these two factors, Quorum proves to be highly suitable and applicable in a variety of practical situations. This result may be consistent with the majority of previous studies, which focused only on eventual consistency to increase performance at the expense of data integrity.

A study by researchers at the National University of Singapore evaluated the impact of different consistency models on the resource utilization of a NoSQL database in distributed environments. The study found that the impact of the consistency model on resource utilization can increase as the number of nodes in the cluster increases. This is because strong consistency models require more coordination and data replication across distributed nodes, which can lead to higher CPU, memory, and network bandwidth consumption.

Overall, proposed model of transaction consistency (PMC) is a promising new approach to achieve the enhancing consistency in *NoSQL* database for read and write transactions. It provides better throughput and latency than MVCC under

different bunch marking strategies, but it does not provide the same strong consistency guarantees this result may consistent with a study of Ogunyadeka et al.[109] in the enhancing of data consistency.

NoSQL workload

The workload of *NoSQL* databases can exert a substantial influence on the performance of the transaction consistency model. In the event that the *NoSQL* database is characterized by a high read-to-write ratio, it is possible that the **PMC model may exhibit superior performance** relative to the MVCC model. This suggests that the specific characteristics of the workload can play a crucial role in determining the efficacy of these transaction consistency models. Therefore, it is essential to carefully consider the workload when evaluating the performance of *NoSQL* databases in order to accurately assess the suitability and effectiveness of different transaction consistency models.

Chapter Seven

Conclusion

6.1 Conclusion

NoSQL databases have been specifically designed to offer flexible and scalable solutions for data storage across a wide array of applications. Nevertheless, a significant challenge within these databases lies in the assurance of consistency and the resolution of conflicts that may arise due to concurrent transactions and the existence of multiple data versions. The objective of this particular study was to introduce the concept of NoSQL transaction consistency (PMC), which aims to provide robust consistency guarantees while simultaneously optimizing the performance of NoSQL databases.

Drawing upon a comprehensive range of experiments, the proposed transaction consistency model (PMC) that was proposed displayed superior transaction consistency when compared to the existing MVCC model. This novel model delivered stronger atomicity guarantees but also enhanced isolation properties, thereby leading to an overall improvement in data integrity and concurrency control. Nevertheless, it is important to note that both models exhibited comparable levels of durability. In order to ensure that concurrent transactions do not impede one another, the suggested NoSQL transaction consistency (PMC) mainly relies on the use of locks. While this approach does provide strong consistency guarantees, it is important to acknowledge that it can potentially result in performance bottlenecks, particularly in environments characterized by high levels of concurrency.

6.2 Recommendations

1. Investigating the use of PMC-Read/Write transactions in conjunction with other concurrency control mechanisms, such as OCC.
2. Developing new ways to measure and quantify the consistency guarantees offered by PMC-Read/Write transactions.
3. Evaluating the performance of PMC-Read/Write transactions in a wider range of workloads and environments.
4. In this study the experiment was conducted on a single database instance. In a production environment, there may be multiple database instances, which can further impact the consistency and performance of transactions.
5. The results of this experiment were obtained on a specific set of hardware and software configurations. The future research may studding the models in configuration of other specific environment
6. The proposed transaction consistency model (PMC) is a newer model, and it is still under active development. This means that the proposed model may have more experiments for improve its performance and scalability in the future.
7. There is ongoing research on how to improve the performance and scalability of the proposed *NoSQL* transaction consistency in other types of *NoSQL* databases. This research could lead to new and innovative approaches to achieving strong consistency for transactions in *NoSQL* databases.
8. The study recommend that conduct more studies for transaction consistency with the relation of availability and throughput with in other data model of *NoSQL* databases like graph model because most of recent studies can't cover this item. Use a smaller replication factor. This will reduce the amount of data that needs to be replicated.

Considerations:

- The experiment should be repeated with different data workloads and different concurrency levels to get a more comprehensive understanding of the performance and accuracy of the two transaction consistency models on the cloud.
- The experiment should also be repeated with different versions of MongoDB to see how the performance and accuracy of the PMC model evolves over time.

Specific Cloud Environment Considerations:

- Instance type: in this study we choose instance types that are appropriate for the workload and the number of concurrent users. For example, if you expect a high concurrency workload, you may want to choose instance types with more cores and memory.
- Networking: the configuration of the networking resources provide high bandwidth and low latency between the VMs.
- Storage: a storage is appropriate for the workload and the size of the database
- Security: an implement appropriate security measures to protect the test environment from unauthorized access.

References

1. A, A.R. *A Novel Approach for Transformation of Data from MySQL to* ***NoSQL*** *(MongoDB)*. 2023 09 23.
2. Liu, A.D.a.M., *Big data systems A software engineering perspective.* ACM Computing Surveys (CSUR), 2020. **53**(5): p. 110-120.
3. DBMS, I.D.a.t. 2023 01 01.
4. Rosenthal., D. *Next gen* ***NoSQL****: The demise of eventual consistency?* 2023.
5. A. Silberschatz, H.F.K., and S. Sudarshan, *Data models.* ACM Computing Surveys (CSUR), 1996. **28**(1): p. 105-108.
6. *URL Domain Modeling*. 2010, Stack Overflow.
7. Ostrovskiy, S., *iOS: Three ways to pass data from Model to Controller.* Medium. A Medium Corporation, 2017.
8. Apache, *Apache Lucene*. 2022, Apache.
9. Massé, M., *REST API design rulebook: Designing consistent RESTful Web Services*. 2012: O'Reilly,.
10. F. Chang, J.D., S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *BigTable: A distributed storage system for structured data.* ACM Transactions on Computer Systems (TOCS), 2008. **26**(2): p. 00-00.

11. Malik, A.L.a.P., *Cassandra: A decentralized structured storage system.* ACM SIGOPS Operating Systems Review, 2010. **44**(2): p. 35-40.

12. P. O'Neil, E.C., D. Gawlick, and E. O'Neil, *The log-structured merge-tree(LSM-tree).* Acta Informatica, 1996. **33**(4): p. 351-385.

13. Ghemawat, J.D.a.S., *MapReduce: Simplified data processing on large clusters.* Communications of the ACM, 2008. **51**(1): p. 107-113.

14. J. C. Corbett, J.D., M. Epstein, A. Fikes, C. Frost, J. J. Furman,S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, and others,, *Spanner: Google's globally distributed database.* ACM Transactions on Computer Systems (TOCS), 2013. **31**(3): p. 8-13.

15. Stonebraker, M., *Why enterprises are uninterested in **NoSQL***. 2011: Welly.

16. Maier, D. *Why database languages are a bad idea*. in *Proceedings of the International Workshop on Database Programming Languages*. 1989.

17. C. Zaniolo, H.A.t.K., D. Beech, S. Cammarata, L. Kerschberg, and D. Maier, *Object-oriented database and knowledge systems*. 1985, Technical ReportDB-038-85, MCC.

18. Liu, K.L.a.L., *Scaling queries over big RDF graphs with semantic hash partitioning.* ACM, 2013. **6**(14): p. 1894-1905.

19. T. Berners-Lee, J.H., O. Lassila, and others, *The semantic web.* Scientific american, 2001. **284**(5): p. 28-37.

20. A. Schenker, A.K., H. Bunke, and M. Last,, *Graph-theoretic techniques for web content mining*. 2005. **62**(1): p. 00-00.

21. Knisley, D.K.a.J., *Graph theoretic models in chemistry and molecular biology*. 2007: Welly.

22. Goodman, J.B.R.a.N. *A survey of research and development in distributed database management*. in *the 3rd International Conference on Very Large Databases*. 1977. VLDB Endowment.

23. Brewer, A.F.a.E.A. *Harvest, yield, and scalable tolerant systems*. in *the 7th Workshop on Hot Topics in Operating Systems*. 1999. IEEE.

24. Brewer, E.A., *Towards robust distributed systems,*. PODC, 2000. **7**.

25. Lynch, S.G.a.N., *Brewer's conjecture and the feasibility of consistent available, partition-tolerant web services.* ACM SIGACT News, 2002. **33**(2): p. 51-59.

26. A. Davoudian, L.C., and M. Liu, *A survey on **NoSQL** stores.* ACM Computing Surveys (CSUR), 2018. **51**(2): p. 40-44.

27. Cattell, R., *Scalable SQL and **NoSQL** data stores.* ACM SIGMOD Record, 2011. **39**: p. 12-27.

28. CHEN, A.D.a.L., *Stores, A Survey on **NoSQL**.* ACM COMPUTING SURVEY, 2018. **51**(2): p. 40-63.

29. Neha Bansal, S.S.L.K.A., *Are **NoSQL** Databases Affected by Schema?* IETE Journal of Research, 2023: p. 1-22.

30. *A quorum-based data consistency approach for non-relational database.* Cluster Computing, 2022.

31. A. Karpenko, O.T., A. Gorbenko, *Research consistency and perfomance of **NoSQL** replicated databases.* Advanced Information Systems, 2021.

32. Sidi Mohamed Beillahi, A.B., Constantin Enea, *Checking Robustness Between Weak Transactional Consistency Models.* ESOP.

33. Adam Krechowicz, S.D., Grzegorz Łukawski, *Highly Scalable Distributed Architecture for **NoSQL** Datastore Supporting Strong Consistency.* Computer Science IEEE Access, 2021.

34. Anatoliy Gorbenko, A.R., Olga Tarasyuk, *Interplaying Cassandra **NoSQL** Consistency and Performance: A Benchmarking Approach.* EDCC Workshops, 2020.

35. Chenggang Wu, V.S., J. Hellerstein. *Transactional Causal Consistency for Serverless Computing*. in *SIGMOD Conference*. 2020.

36. Ranadeep Biswas, C.E., *On the complexity of checking transactional consistency.* Proc. ACM Program. Lang., 2019.

37. Shale Xiong, A.C., Azalea Raad, P. Gardner, *Data Consistency in Transactional Storage Systems: a Centralised Approach.* ArXiv, 2019.

38. María Teresa González-Aparicio, M.Y., Javier Tuya, Ruben Casado, *Testing of transactional services in* ***NoSQL*** *key-value databases.* Future Gener. Comput. Syst.

39. R. Jiménez-Peris, M.P.-M., Ivan Brondino, V. Vianello, *Transaction management across data stores.* International Journal of High Performance Computing and Networking, 2018.

40. Faour, N., *Data Consistency Simulation Tool for* ***NoSQL*** *Database Systems.* arXiv.org, 2018.

41. Xiangdong Huang, J.W., P. Yu, Jian Bai, Jinrui Zhang, *An experimental study on tuning the consistency of* ***NoSQL*** *systems.* Concurrency and Computation, 2017: p. 16-28.

42. María Teresa González-Aparicio, A.O., Muhammad Younas, Javier Tuya, Ruben Casado, *Transaction processing in consistency-aware user's applications deployed on* ***NoSQL*** *databases.* Human-centric Computing and Information Sciences, 2017.

43. A. Burdakov, Y.G., A. Ploutenko, Eugene Ttsviashchenko. *Estimation Models for* ***NoSQL*** *Database Consistency Characteristics*. in *International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 2016.

44. Adewole Ogunyadeka, M.Y., Hong Zhu and A. Aldea. *A Multi-key Transactions Model for* ***NoSQL*** *Cloud Database Systems*. in *2016 IEEE Second*

International Conference on Big Data Computing Service and Applications (BigDataService) (2016). 2016.

45. Lotfy, A., Saleh, A., El-Ghareeb, H., & Ali, H., *A middle layer solution to support ACID properties for **NoSQL** databases.* J. King Saud Univ. Comput. Inf. Sci.,, 2016. **28**: p. 133-145.

46. Adewole Ogunyadeka, M.Y., Hong Zhu, A. Aldea. *A Multi-key Transactions Model for **NoSQL** Cloud Database Systems*. in *International Conference on Big Data Computing Service and Applications*. 2016.

47. Madhavamuniappan, ***NoSQL** Concatenated Transactions for Numerous Application in the Cloud.* Computer Science, 2014.

48. Islam M, V.S., *Comparison of consistency approaches for cloud databases.* International Journal Of Cloud Computing 2013. **2**(4): p. 378-398. .

49. Wada H, F.A., Zhao L, Lee K, Liu A. *Data Consistency Properties and the Trade- offs in Commercial Cloud Storages: the Consumers' Perspective*. in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research*. 2011. Asilomar, CA, USA.

50. Kraska T, H.M., Alonso G, Kossmann D. *Consistency rationing in the cloud: Pay only when it matters*. in *Proceedings Of The VLDB Endowment*. 2009.

51. A. Dey, A.F., and U. Röhm. *Scalable transactions across heterogeneous **NoSQL** key-value data stores*. in *Proceedings of the VLDB Endowment*. 2013.

52. Coelho FACL, C.F.d., Vilaca RMP, Pereira JO, Oliveira, *pH1: A Transactional Middleware for **NoSQL***. IEEE 33rd International Symposium on Reliable Distributed Systems Available, 2014.

53. M. A. Mohamed, O.G.A., and M. O. Ismail,, *Relational vs. **NoSQL** databases: A survey.* International Journal of Computer and Information Technology, 2014. **3**(3): p. 598-601.

54. Paz, J.R.G., *Introduction to azure cosmos db*, in *Microsoft Azure Cosmos DB Revealed: A Multi-Model Database Designed for the Cloud*. 2018, Berkeley, CA: Apress. p. 1-23.

55. L. Perkins, E.R., and J. Wilson, *Seven databases in seven weeks: a guide to modern databases and the **NoSQL** movement*. 2018: Pragmatic Bookshelf,.

56. G. Haughian, R.O., and W. J. Knottenbelt, *Benchmarking replication in cassandra and mongodb **NoSQL** datastores*, in *Database and Expert Systems Applications*. 2016, Cham: Springer International Publishing,. p. 152-166.

57. Vogels, W., 52(1), 40-44., *Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.* Communications of the ACM, 2009. **52**(1): p. 40-44.

58. Ghodsi, P.B.a.A., *Eventual consistency today: Limitations, extensions, and beyond.* Queue, 2013. **11**: p. 03-20.

59. Viotti, P., and Marko Vukolić., *Consistency in non-transactional distributed storage systems.* ACM Computing Surveys (CSUR), 2016. **49**(1): p. 1-34.

60. Jepsen. *Monotonic Reads.* 2022.

61. Writes, J.M.

62. Chihoub, H.E., *Managing Consistency for Big Data Applications on Clouds: Tradeoffs and Self-Adaptiveness*, in *THÈSE / ENS CACHAN - BRETAGNE.* 2013.

63. María Teresa González‑Aparicio, A.O., Muhammad Younas, Javier Tuya, *Transaction processing in consistency‑aware user's applications deployed on **NoSQL** databases.* Human Centric Computer Information System, 2017. **7**(7): p. 2-18.

64. S. P. Kumar, S.L., R. Chiky, and O. Hermant, *Consistencylatency trade-off of the libre protocol: A detailed study.* Advances in Knowledge Discovery and Management, 2018. **7**: p. 83-108.

65. E. Casalicchio, L.L., and S. Shirinbab, *Energy-aware autoscaling algorithms for cassandra virtual data centers.* Cluster Computing, 2017. **20**(3): p. 2065-2082.

66. Harrison, G., *Consistency models*, in *Next Generation Databases: **NoSQL**, NewSQL and Big Data*. 2015, Berkeley, CA: Apress. p. 127-144.

67. Yahoo *Yahoo Cloud Service Benchmark*. 2010.

68. ScyllaDB *ScyllaDB Documentation - Consistency*. 2023.

69. Charvet F, P.A. *Database Performance Study*. 2016.

70. Islam M, V.S., *Comparison of consistency approaches for cloud databases.* International Journal Of Cloud Computing, 2013. **2**(4): p. 378-398.

71. D., G.C., *BASE analysis of* ***NoSQL*** *database. Future Generation Computer Systems* Cloud Computing: Security, Privacy and Practice, 2015. **52**: p. 13-21.

72. Jimenez-Peris, R.V.R. *Maximizing Quorum Availability in Multi-Clustered Systems*. in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing*. 2008. Toronto, Canada.

73. Rouse, M. *Data Replication*. 2013.

74. Anand A, M.C., Akella A, Ramjee R., *Redundancy in network traffic:findings and implications.* Performance Evaluation Review, 2009. **37**(1): p. 37-48.

75. Wang Z, S.K., Jajodia S. *Verification of Data Redundancy in Cloud Storage*. in *CloudComputing '13: Proceedings of the 2013 international workshop on Security in cloud computing*. 2013.

76. Shuyi C, J.K., Hiltunen M, Schlichting R, Sanders W., *Using Link Gradients to Predict the Impact of Network Latency on Multitier Applications.* IEEE/ACM Transactions On Networking, 2011. **19**(3): p. 11-27.

77. Rouse, M. *Latency*. 2023.

78. Team, M. *Data Consistency Primer*. 2015.

79. Takahiro Yasui, Y.I., Tomohito Ikedo. *Influences of network latency and packet loss on consistency in networked racing games*. in *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. 2015.

80. P. A. Bernstein, V.H., and N. Goodman, *Concurrency Control and Recovery in Database Systems*. 1986, Boston, MA, USA: Addison-Wesley.

81. R. Smith, R.H., S. Wood, D. Sussman, A. Fedorov, S. Murphy, and Home, *Professional Active Server Pages 2.0*. 1998, Birmingham, UK, UK, UK, UK: Wrox Press Ltd.

82. J.C. Corbett, e.a., *Spanner: Google’s globally distributed database.* ACM Transactions Computer System, 2013. **31**: p. 1-22.

83. W. Lloyd, e.a. *Don't settle for eventual: scalable causal consistency for widearea storage with COPS*. in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011. Cascais, Portugal.

84. J. Cowling, B.L. *Granola: low-overhead distributed transaction coordination,*. in *USENIX ATC'12*. 2012. Boston, MA.

85. R. Escriba, e.a. *Warp: Multi-Key Transactions for Key-Value Stores*. 2013.

86. J. Baker, e.a., *Megastore: Providing Scalable, Highly Available Storage for nteractive Services.* CIDR,, 2011: p. 223-234.

87. S. Das, e.a. *G-Store: a scalable data store for transactional multi key access in the cloud*. in *The 1st ACM symposium on Cloud computing*. 2010. Indianapolis, Indiana, USA.

88. J.J. Levandoski, e.a., *Deuteronomy: Transaction Support for Cloud Data.* CIDR, 2011: p. 123-133.

89. K. Chitra, B.J., *Cloud TPS: Scalable transaction in the cloud computing.* International Journald of Engineering and. Computer Science, 2013. **2**: p. 2280-2285.

90. F. Coelho, e.a. *pH1: middleware transacional para* ***NoSQL***. in *IEEE 33rd International Symposium on Reliable Distributed Systems SRDS*. 2014.

91. R. Jimenez-Peris, e.a., *CumuloNimbo: A cloud scalable multi-tier SQL database,.* IEEE Data Engineerinh. , 2015. **38**: p. 73-83.

92. A.E. Lotfy, e.a., *A middle layer solution to support ACID properties for* ***NoSQL*** *databases,.* Journal of King Saud University for Computer Infastructure Science 2016. **28**: p. 133–145.

93. V. Padhye, A.T., *Scalable transaction management with snapshot isolation for* ***NoSQL*** *data storage systems.* IEEE Transactions Survey Computer, 2015. **8**: p. 121-135.

94. F. Junqueira, e.a. *Lock-free transactional support for large-scale storage systems*. in *Dependable Systems and Networks Workshops, DSN-W, 2011 IEEE/IFIP 41st International Conference on* 2011.

95. D. Peng, F.D., , in:, *Large-scale incremental processing using distributed transactions and notifications.* OSDI, 2010: p. 1-15.

96. V. Vafeiadis, e.a. *Proving correctness of highly-concurrent linearisable objects*. in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2006.

97. M.P. Herlihy, J.M.W., *Linearizability: A correctness condition for concurrent objects.* ACM Transaction Programing Langauges System, 1990. **12**: p. 463-492.

98. Adya., A., *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* 1999, PhD thesis, MIT, Cambridge: MA, USA, March 1999.

99. K. P. Eswaran, J.N.G., R. A. Lorie, and I. L. Traiger, *The notions of consistency and predicate locks* ACM database system. Commun. , 1976. **19**(11): p. 624-633.

100. Goodman., P.A.B.a.N., *Concurrency control in distributed database systems.* ACM Computer Survey, 1981. **13**(2): p. 185-221.

101. Michael J. Cahill, U.R., and Alan D. Fekete, *Serializable isolation for snapshot databases.* ACM Transactions on Database Systems, 2009. **34**(4): p. 1-42.

102. Thomas Neumann, T.M., and Alfons Kemper. *Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems*. in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD ’15*. 2015.

103. Weisberg, M.S.a.A., *The VoltDB Main Memory DBMS.* IEEE Data Eng. Bull, 2013: p. 21-27.

104. al., H.B.e. *A critique of ANSI SQL isolation levels*. in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIG-MOD ’95*. 1995. New York, New York, USA: ACM PRESS.

105. S. A. Weil, S.A.B., E. L. Miller, D. D. E. Long, and C. Maltzahn. *Ceph: A scalable, high-performance distributed file system*. in *OSDI*. 2006.

106. Peter Bailis, A.F., Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica, *Coordination avoidance in database systems.* Proc. VLDB Endow., 2014. **8**(3): p. 185-196.

107. Peter Bailis, A.F., Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. *Scalable atomic visibility with RAMP transactions*. in *In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD ’14*. 2014. New York, NY, USA.

108. María Teresa González-Aparicio, A.O., Muhammad Younas, Javier Tuya, Rubén Casado, *Transaction processing in consistency-aware user's applications deployed on **NoSQL** databases.* Human-centric Computing and Information Sciences, 2017. **7**(12): p. 1-18.

109. A. Ogunyadeka, M.Y., H. Zhu and A. Aldea. *A Multi-key Transactions Model for **NoSQL** Cloud Database Systems*. in *IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*. 2016. Oxford, UK.

Appendix (1)

The (PMC) Model Source Code in MongoDB

```
# Define the database system and the operations

db = MongoDB("mongodb://localhost:27017")

coll = db["test"]["items"]

ops = [ {"insertOne": {"document": {"_id": 1, "name": "apple", "price": 0.99}}},
{"updateOne": {"filter": {"_id": 2}, "update": {"$set": {"price": 1.99}}, "upsert":
True}}, {"deleteOne": {"filter": {"_id": 3}}}]

# Specify the transaction options

rc = ReadConcern("snapshot")

wc = WriteConcern("majority")

rp = ReadPreference("primary")

to = 10 # seconds

# Start a session and a transaction

session = db.startSession()

session.startTransaction(readConcern=rc, writeConcern=wc, readPreference=rp)
```

```
# Execute the operations in the transaction

try:

  result = coll.bulkWrite(ops, session=session)

  print(result)

except Exception as e:

  print(e)

  session.abortTransaction()

  session.endSession()

  return

# Commit or abort the transaction

if result.ok:

  try:

    session.commitTransaction()

    print("Transaction committed")

  except Exception as e:

    print(e)

    session.abortTransaction()
```

```
    print("Transaction aborted")

else:

  session.abortTransaction()

  print("Transaction aborted")

session.endSession()
```

Appendix (2)

Experiments Test code with python

```
import pandas as pd

# Experiment 1: Scalability

experiment_1_results = pd.DataFrame({
    "Workload (transactions/second)": [100, 200, 400, 800],
    "PMC Latency (ms)": [10, 15, 25, 40],
    "MVCC Latency (ms)": [20, 30, 50, 80],
    "PMC Throughput (transactions/second)": [120, 220, 380, 650],
    "MVCC Throughput (transactions/second)": [100, 180, 300, 500]
})

print("Experiment 1: Scalability")
print(experiment_1_results)

# Experiment 2: Consistency

experiment_2_results = pd.DataFrame({
    "Failure Scenario": ["Node failure", "Network partition",
"Data corruption"],
    "PMC Consistency": ["Maintained", "Maintained",
"Maintained"],
    "MVCC Consistency": ["Maintained", "Violated", "Maintained"]
})

print("\nExperiment 2: Consistency")
print(experiment_2_results)

# Experiment 3: Performance

experiment_3_results = pd.DataFrame({
    "Workload (transactions/second)": [100, 200, 400, 800],
    "PMC Latency (ms)": [5, 10, 15, 20],
    "MVCC Latency (ms)": [10, 20, 30, 40],
    "PMC Throughput (transactions/second)": [150, 250, 400, 600],
    "MVCC Throughput (transactions/second)": [120, 200, 350, 550]
})

print("\nExperiment 3: Performance")
print(experiment_3_results)
```

بسم الله الرحمن الرحيم

جامعة شندي

Shendi University

عمادة البحث العلمي

Deanship of Scientific Research

مجلة جامعة شندي للعلوم التطبيقية

Shendi University Journal of Applied Science

ISSN: 1858-9022

شـــهـــادة نشـــر

Publishing Certificate

بهذا تشهد مجلة جامعة شندي للعلوم التطبيقية بأن الموضوع الموسوم بالعنوان التالي:

AN EVALUATION OF CONSISTENCY MODELS IN NOSQL DATABASES

والذي تقدم به الباحث

محمد صديق حسن عبدالقادر

قد تم نشره في عدد المجلة العاشر (1-2023) والذي تم إصداره في تاريخ يونيو 2023م.

حررت هذه الشهادة في 2023/12/25 ص 7:43:59

لتقديمها لمن يهمهم الأمر

د. وسيم سمير كوامي سليمان

رئيس التحرير