Consistency Issues on NoSQL Databases: Problems with Current Approaches and Possible Solution(s)

Jabir Al Fatah
Authors
Jabir Al Fatah

Title
Consistency Issues on NoSQL Databases: Problems with Current Approaches and Possible Solution(s)

Supervisor
Christian Andersson

Examiner
Dawit Mengistu

Abstract
NoSQL database is the answer to the problem of scalability with traditional relational databases. Due to the high volume of data across distributed platforms at different geographical locations, non-relational databases such as NoSQL has got popularity among big companies and enterprises. Yet, the limitation to maintain consistency among these data sources has become an issue.

This degree project deals with data inconsistency issues on distributed NoSQL databases and currently used approaches to maintain consistency. The research work discusses some key factors behind data inconsistency in distributed NoSQL databases. For this thesis, BASE and Quorum are considered, which are two of the available techniques to maintain data consistency in distributed architectures. This paper analyzes their satisfactory level of consistency and identify the Quorum that needs to be further optimized.

Furthermore, a suggestion is proposed on how to improve the Quorum protocol based on some experimentations and observations.

Keywords
Data inconsistency, NoSQL, Cassandra, Riak, BASE, Quorum, ACID, YCSB.
**Table of Contents**

<table>
<thead>
<tr>
<th>Document page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document page</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iii</td>
</tr>
</tbody>
</table>

1 **Introduction** 5
1.1 Background 5
1.2 Research Questions 5
1.3 Motivation 6
1.4 Consistency Approaches 7

2 **Methodology** 12
2.1 Method Description 12
2.2 Research Area 12
2.3 YCSB: Database Benchmarking Tool 19

3 **Results** 23
3.1 Data Inconsistency 23
3.2 Satisfactory Level of the Approaches 25
3.3 Optimization of Quorum 29

4 **Discussion** 33
4.1 Data Inconsistency Factors 33
4.2 Satisfactory Level of the Approaches 33
4.3 Suggestion of Quorum Optimization 34

5 **Conclusion** 35

6 **Future Works** 36
6.1 Finding More Inconsistency Reasons 36
6.2 Measure Consistency Level with Other NoSQL Databases 36

7 Bibliography 37
1 Introduction

1.1 Background

Every web application and service uses data. Modern web and mobile applications are designed to support large number of concurrent users by spreading load across multiple computers \(^1\). The traditional relational databases (i.e., MySQL) are facing many problems to contend with these trends due to their dependency on fixed schema and inability to scale well \(^1\).

NoSQL databases, on the other hand, were explicitly designed to handle this large scale needs and storage capacity limitations of relational databases. NoSQL (Not Only SQL) is a non-relational, distributed and horizontal scalable database. Unlike relational databases, a vast majority of NoSQL do not provide the guarantee of data consistency, although it is one of the important property. Data consistency property implies that all instances of an application are presented with the same set of data values all the time. Data inconsistency exists when different and conflicting versions of the same data appear in different places \(^2\). Inconsistency creates unreliable information, because it will be difficult to determine which version of the information is correct \(^2\).

1.2 Research Questions

In order to narrow down the horizon of consistency problem and address some specific principles within this topic, the following research problems have been formulated.

**Research question 1: What are the main factors that cause data inconsistency in distributed applications and data?**

Data inconsistency is very common drawback of distributed applications. In fact, people face this issue every now and then while making any online transaction. Applications with distributed nature such as cloud applications are likely to be partitioned across data centres hosted at different geographical locations. This can occur for numerous reasons, for example, to improve availability by replicating data and to improve scalability by balancing the load across multiple servers. However, maintaining and managing data consistency in this environment has become a critical aspect of the system. The first part of the thesis will identify the conditions that create the inconsistency state in distributed applications and data.

**Research question 2: How satisfactory is the performance of current approaches to maintain consistency on NoSQL databases?**

There are few available approaches such as BASE, Quorum and CAP that are used to maintain the consistency of NoSQL databases \(^7\), \(^19\), \(^22\). These consistency approaches have been established based on application requirements, and or, technical capabilities. In this part of the research, the quality of the consistency performances of BASE and Quorum will be demonstrated. Two NoSQL databases, Cassandra (version 3.7.0) and Riak (version 2.1.4) are selected for consistency analysis those use BASE and Quorum approach respectively, to maintain their consistency. Although there is a wide variety of NoSQL databases, it is practically impossible to discuss and fit all of their working approach and consistency in this research project. Hence, some of the popular NoSQL (i.e., Cassandra, Riak) are studied. I expect to measure the satisfactory level of these approaches by benchmarking Cassandra and Riak.
Research question 3: How current approaches can be optimized to get better performance?

NoSQL databases are new technology in modern computing era and this solution is built for 21st century’s distributed environment. The technology is still immature for many industrial operations. Currently available approaches for maintaining consistency on NoSQL is not yet very advanced. This part of the work will focus on optimization of one of the approach as well as finding ways for its better implementation.

1.3 Motivation

The consistency maintenance has become the primary obstacle to the implementation of transactional data management system in distributed platforms, while a vast majority of computer industries and users are migrating to NoSQL solutions. The companies (e.g., Google, Facebook, and Amazon) are increasingly depending on big data and real-time web applications. Nowadays, strong consistency requirement has become an imperative concern for many notable web-scale applications.

Leading NoSQL databases such as Cassandra, Couchbase, and DynamoDB provide clients applications with guarantee of eventual consistency rather than immediate consistency \[^{[5]}\]. Developing distributed application for distributed system with eventual consistency takes a lot of developers’ time. Additionally, this weak feature forces developers to do the hard work to ensure that multiple clients do not step on each other’s toes and deal with stale data \[^{[5]}\]. In fact, eventual consistency represents a dramatic weakening of the guarantees that traditional databases provide and places a huge burden on software developers \[^{[5]}\], though it is also not the worth effort to ensure consistency of every type of data (e.g., Facebook and search cache).

Developing the advanced, scalable system demanded by today’s highly connected world with such poor guarantee is exceptionally challenging. For software developers, it is essential to stop accepting weak consistency and explore scalable, distributed database design that offer more reliable data consistency. So there is an obvious need to inspect the factors that are causing the inconsistency.

NoSQL database is a recent effort and I have seen the rise of this new type of database over the last few years. So the technology is still new to many database designers. Few examples are:

- A popular document based NoSQL database called MongoDB has started in 2007 and its first standalone release was in 2009.
- A scalable elastic document store, Terrastore, started in 2009.

That makes me to believe that working with performance enhancement of this newly emerged technology is a worth effort spending. Therefore, for application developers, it is important to take the performance factors into consideration. Some efforts regarding database performance issue are as follow:

Islam Md. A, Vrbsky S V, \[^{[7]}\] has demonstrated the advantage and weakness of few consistency techniques. There was no method introduced for performance optimization in this article.
Kraska T, Hentschel M, Alonso G, Kossmann D. [8] developed new transaction paradigm for consistency guarantee, but nowhere in the paper have they shown the satisfactory level of performance of different consistency approach such as BASE and Quorum.

Wada H, Fekete A, Zhao L, Lee K, Liu A. [9] have investigated what consumers observe of the consistency and performance properties of various offerings. The writers have not given any clue regarding the optimization process of different approaches. The missing endeavours within the current research works convince me to contribute more on the performance matter of consistency approaches on NoSQL as well as possible suggestion to improve them.

1.4 Consistency Approaches

In order to maintain consistency, there are limited number of approaches that have been proposed. As mentioned earlier, the limitation of these available approaches are that they do not maintain full consistency. With the increasing amount of data being processed by companies’ information systems, database performance issues become more common. In the meantime, user requirements and expectations are constantly on the rise, and delay in response time could considerably affect the company’s operations [6]. Thus, much time should be spent to measure the satisfactory level of performance of currently used approaches. Not to mention, approaches like Quorum and BASE are two of the mostly used property followed by many popular NoSQL database solutions. Some significantly important NoSQL databases such as Dynamo, GFS (Google file system), BigTable, Cassandra, Hadoop DFS, Haystack follow BASE [10]. BASE is also popular among database designers because: if users are partitioned across five data centres, BASE design encourages crafting operations in such a way that a user database failure impacts only the 20% of the users on that particular host [10]. Quorum, however, is one of the oldest technique that has long been used in distributed data storage systems. Quorum has gained its high popularity among developers because the system is symmetric. Symmetry implies that the quorum system is uniform and fair (i.e., all quorums are of the same size and the load is evenly distributed across the sites) [42]. We, therefore, have decided Quorum and BASE to be studied in this project as both of them are inevitably influential approach used in modern distributed system.

1.4.1. BASE

BASE (Basically Available, Soft state, Eventual consistency) property is an eventual consistency model that makes sure that data in a store is eventually consistent. The property does not require the database to be consistent after every transaction. BASE was proposed by eBay. Researchers at Amazon have also adopted BASE. eBay services run in a single data centre. Their data centres are reliable, deliberately engineered to use potentially incorrect data [10]. According to Chandra G D [10], “BASE is a database design philosophy that prizes Availability over Consistency of operations. BASE ensures at least 80% consistency at any given instance over the flow of the operation”.

BASE is:

Basically Available: It means data in a database system is available most of the time. In this system data loss can occur because of system failure. And only catastrophic failure can make the whole system down.
The availability of BASE is achieved through supporting partial failures without total system failure. There is no magic involved, but this does lead to higher perceived availability of the system\textsuperscript{[11]}.

**Soft state:** This state provides relaxed view of data in terms of consistency. With soft state, the data copies may be inconsistent and the state of the system may change over time, even without input because of the “Eventual Consistency\textsuperscript{[10]}”. While soft state is made unavailable due to the crashes in service instance, reconstructing it through user interaction can be expensive in terms of times and resources\textsuperscript{[10, 12]}.

**Eventual consistency:** Eventual consistency does not require all the nodes in a network system to have identical copies of data all the time. The update in the replicas will only propagate when there are no updates over a certain period of time for a particular application\textsuperscript{[10]}. Although eventual consistency is widely deployed in many distributed systems, it is also obvious that the provided guarantee by this consistency is weak. Eventual consistency is also responsible for increasing the complexity of distributed applications.

The discussion below is about the two mostly used NoSQL databases that are designed with the principle of BASE.

1. **The Apache Cassandra** is an open source distributed database. Cassandra is the right choice for many cloud companies because of its linear scalability and proven fault-tolerance on commodity hardware\textsuperscript{[40]}. The database was initially developed by Facebook to power the Facebook inbox search feature\textsuperscript{[41]}. More than 1500 web companies, including eBay, Instagram, Netflix and Reddit, are constantly using Cassandra’s service without any network bottleneck and any single point of failure\textsuperscript{[40]}. Cassandra is highly scalable and incredibly elastic. For instance, some of the largest production deployments include Apple’s, with over 75,000 nodes storing over 10 PB of data, Netflix (2,500 nodes, 420 TB), Chinese search engine Easou (270 nodes, 300 TB, over 800 million requests per day), and eBay (over 100 nodes, 250 TB)\textsuperscript{[40]}. Additionally, Cassandra service is available from many third parties such as DataStax, Cubet and Workware system, that makes it possible to run the data store on almost any operating system. Another key point is that, to benchmark Cassandra, it does not require any dependencies, and the popular benchmarking tool YCSB provides varieties of Cassandra binding to examine its consistency.

2. **Dynamo** is a distributed data storage system that uses eventual consistency model. It powers Web services like Amazon S3 for cloud computing, including Dropbox and Ubuntu one\textsuperscript{[10, 13]}, Dynamo uses self-repair mechanism to implement eventual consistency in practice\textsuperscript{[10]}.

### 1.4.2. Quorum

Quorum approach is a voting system where the decision will be made when the maximum number of nodes distributed platform agree with particular decision. As described by Islam Md. A, Vrbsky S V\textsuperscript{[7]}, “a quorum-based consistency system requires more than half of the replicas to complete a read or write operation before the data is available for a subsequent read or write”. Quorum-based technique does not require any specific primary and secondary replica to perform operation. In fact, any replica is allowed to act as the primary node, sending read and write operation to any other node\textsuperscript{[7]}.
As shown in Figure 1(a), a write request is allocated from primary node to every other node with a replica of the data. Once the maximum number of nodes send back the acknowledgement, the data becomes available for read and write. Similarly, in Figure 1(b), when most of the nodes return the updated data to the primary key, the write becomes available for proceeding. However, the quorum approach is unable to maintain full consistency. It provides eventual consistency, without sacrificing too much performance.

The discussion below is about the two widely used databases that follow Quorum technique.

1) **Riak**, is remarkably scalable NoSQL data store that is highly available and easy to operate. Riak was developed by Basho and it was written in Erlang. In Riak, objects are stored and retrieved in JSON (JavaScript Object Notation) format and can have multiple fields and can be grouped into buckets (collections in document databases). It is currently used by Github, Comcast, Voxer, Disqus, Yahoo Japan and others, with the larger systems storing hundreds of TBs of data, and handling several GBs per node every day. There is a great deal of reasons that influence developers to adopt Riak among many NoSQLs. Like Cassandra, Riak data stores are masterless, able to excel at high write volume and highly available stores that persist replicas and handle failure scenarios through concepts such as hinted handoff and read-repair. Riak is enabled with vector clocks, a data structure, that tracks its causal ordering of updates. This per-object ancestry allows the data store to identify and isolate conflicts without using system clocks.

To handle divergent and lost data, Riak provides read-repair mechanism where read operation triggers the inconsistent and missing data automatically becomes repaired in the background. Most importantly, Riak is good at dealing with data
availability with as few downsides as possible. The database is designed to survive network partitions and hardware failures that would significantly disrupt most databases \[39\]. The Riak key/value store is built with remarkable facility. For example, it is able to write data even when a client can reach only one Riak server, and the datacentre provides multi-datacentres cluster replication, which ensures the robustness. Furthermore, Riak’s official site serves a large number of documentation for building and maintaining its foundation for distributed application on enterprise level. With all those points being noted, Riak is certainly a good selection to benchmark, among many other databases.

However, Riak, by default, uses Quorum-based approach to maintain consistency. A quorum in Riak can be expressed as $N/2+1$, where $N$ is the number of participating nodes. If $N=5$, 3 replicas constitutes a quorum; if $N=7$, 4 replicas, etc. If $N=5$ and 3 replicas are unavailable, no strongly consistent operations will be succeeded. When Riak is used as an eventually consistent system, the data that to be read should remain available in most failure scenarios, although it may not be the most up-to-date version of that data \[39\]. Thus, the consistency remains eventual for the data store.

2) **Azure DocumentDB**, is a data-storage system running on cloud-based service, provided by Microsoft Corporation. The official website claims that DocumentDB guarantees less than 10 ms latencies on reads while 15 ms on writes for 99% of requests \[17\]. Tables of this database offer NoSQL capabilities at a low cost for applications with simple data access needs \[17\]. It maintains at least three copies of data and stores each copy on a different physical server \[18\].

1.4.3. **Use Case**

Despite the fact that ensuring stronger consistency is great need for distributed application, not all of them should necessarily have strong consistency maintenance system. Therefore, this research brings to light only those applications where NoSQL database is a necessity, and immediate consistency is not a higher priority but the availability. So, our particular interest is to relate those areas where sacrificing strong consistency is somehow acceptable.

Interestingly enough, it has not escaped our notice that for many applications, NoSQL is less required even though they handle large quantities of data and users. Some ideal examples \[43\]: video sharing site YouTube, question-and-answer site Quora, and social networking service Twitter are relying on relational MySQL database since they are prioritizing to update the replicas as fastest possible time, or in another words, maintaining consistency and durability. There is more: many banking applications and financial organizations heavily depend on RDMS, because, many of them does not need to handle unstructured immense size of data, and guaranteeing the quick update in all replicas is a necessity. As mentioned in Oracle’s case studies \[44\] in the official site, PayPal powers its GFDS (Global Fraud Detection System) with MySQL Cluster, and the system has been designed to cater for PayPal's high growth and extreme uptime requirements, demanding: 99.999% availability without data loss, ACID-compliance, transactional support, and ability to write data to the database and read it anywhere in the world in under one second.

However, back to our particular use case for this research, the type of distributed applications that has been considered are social communication platforms like Facebook, Google+, LinkedIn; streaming media such as Spotify, Netflix etc. So, the investigation about the
responsible factors for data inconsistency, and problems with current approaches imply only those aforementioned type of distributed applications.
2 Methodology

2.1 Method Description

To discover the factors that cause data inconsistency in distributed applications, a meta-analysis method, which involves analysis and combination of multiple scientific studies, has been followed. Related literatures were systematically reviewed, and then synthesised the results. However, academically authoritative articles, journals, course literature or recommended books, research materials, text on trusted online sites were prioritised 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 review were conducted to extract data comprehensively that relates the first research problem.

This thesis work concentrates on two widely used approaches to be analysed; BASE and Quorum. The analysis includes reading and examining articles and documents on those approaches to identify the process and outcomes.

Furthermore, the satisfactory level of the consistency performance was realized through the practical experiment conducted using YCSB benchmarking tool. The details of YCSB system architecture is presented in section 2.3.1.

In order to show the performance level, the 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 I 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 performance 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 level of the respective 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 Riak 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.

2.2 Research Area

To answer the stated research question, it is relevant to focus on the areas below to explore the scope of database system in a distributed world:
2.2.1 NoSQL Database

NoSQL database is a newly emerged database to deal with massive amount of data across distributed servers. NoSQL is defined as not only SQL; some NoSQL also supports SQL-like query language. Currently there are more than 150 NoSQL databases exist in the market[10]. Some of the big contribution of NoSQL databases are[10]:

- Google’s Big Table, started by Google in 2004
- Amazon’s Highly Available Key-value Store: Dynamo

In most cases, NoSQL databases partition and replicate their data across multiple nodes. This causes read-write conflict in distributed systems. Conflicts occur when some nodes receive updated value and others do not.

The reason that most of the NoSQL databases fail to maintain immediate consistency is because they do not comply with ACID properties, instead they follow BASE. However, although many NoSQL databases include features that can reduce the likelihood of inconsistent data, in many cases they do not provide any viable solution. They have many constraints such as their provided guarantees for transactions are very week and they do not offer easy abstraction to the application programmers. For example, in BASE, exact values are not utterly necessary, and its design principle prizes availability over consistency of operation[10]. Although eventual consistency seems to work “well enough” in practice[3], this type of consistency pushes the pain and perplexity of inconsistent reads and unreliable writes onto application designers. Google addressed the pain points of eventual consistency in a recent paper on its F1 database and noted[4]:

“We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level”.

More details about these properties are discussed in section 2.2.3.

However, the poor performance and weak consistency of NoSQL is observed when the system is deployed in wide geographical datacentres in distributed environment. Such environment requires communication through cross-datacentres messages, and the servers in cross-datacentres are most likely to cause delay while passing messages among different nodes. Thus, the large cross-data communication delay is one of the most dominant reasons, which causes weak performance of most algorithms achieving strong consistency in distributed application[20].
Figure 2. Data size and data complexity of NoSQL databases

Figure 2 shows an overview of four main NoSQL database categories: Key-value store, Column-family store, Document database and Graph database. The different categories represent the different capacities of their data sizes, and their data complexities. It is obvious from the figure that Key-value data stores have adequate capacity to accept very large data size and they are best suitable with data-intensive application. However, a related point to consider is, Graph database is best fit for the applications that deal with complex data.

2.2.2 Mathematical Analysis: Quorum

This section provides some in-depth analytical definitions of quorum based protocol. This representative numerical study is followed by some of the classical quorum systems discussed in literature.

The article “On the trade-off of availability and consistency for quorum systems in data centre networks” written by Xu Wang and his colleagues made an important mathematical argument about maintaining strong consistency by Quorum.

Quorum is actually a set system that is a set of subsets of all replicas \( U \). In practical, quorum systems, two parameters \( W \) and \( R \) are predefined for \( N \) replicas, that is \( N = |U| \) and \( 1 \leq W, R \leq N \). Let’s suppose the write quorum set \( S_{write} = \{ Q | Q \subseteq U \land |Q| = W \} \), and the read quorum set \( S_{read} = \{ Q | Q \subseteq U \land |Q| = R \} \). Obviously, if \( W + R > N \), two elements from \( S_{write} \) and from \( S_{read} \), respectively, will intersect. By doing so, any read can return the newest write and strong consistency is guaranteed.

Literally, in quorum system, the writer timestamps the data and write it to a majority of servers in order to write data to the database. Furthermore, the reader reads data, and that is done through contacting the majority, that eventually return the data with highest timestamp. The majority-intersection property in this setting ensures that the reader will obtain the recent value. However, when the potential network connection comes into present, this become critical in terms of consistency preservation. As discussed in the article, “The Origin of Quorum System”, Quorum can be defined in a more general context, refining the concept of majorities to have arbitrary quorum sizes while maintaining the requirement for non-empty
pair-wise quorum intersections. Starting with the basic definition, or a variation of the
definition\textsuperscript{[45]}:

Given a set \(S = \{s_1, s_2, \ldots, s_n\}\), where \(n \geq 1\), and \(QS\) is a quorum system over \(S\), if and only if

(Intersection) \(\forall Q_1, Q_2 \in QS: Q_1 \cap Q_2 \neq \emptyset\).

Elements in a set of quorum are also called quorum. Once \(S\) is understood, we exclude it for
simplicity. Gifford\textsuperscript{[45, 46]} has divided Quorums into two classes: read and write quorums, and
requires only quorums that belongs to different classes for intersecting. In fact, this
refinement with its distinctions between read and write quorums can be applied to any
quorum system\textsuperscript{[45]}. Although the principle behind quorum construction are same for all
quorum systems, in order to broaden the horizon of our understanding, I briefly instantiate
quorum systems through few classical examples\textsuperscript{[45]}.

Singleton, is one of the simplest quorum system containing a singleton:

\(\text{Singleton} = \{\{s_i\}\}\), where for some \(s_i \in S\).

FPP (Finite projective planes), a kind of quorum system which was used by a researcher in
his mutual exclusion algorithm. If a set \(S\) consists of \(n = k^2 + k + 1\) nodes, where \(k\) is
considered as a prime power, then a FPP of order \(k\) is a quorum system, in which:

- each quorum contains exactly \(k + 1 = O(\sqrt{n})\) nodes
- each node is located in exactly \(k + 1\) quorums, and
- each two quorums have intersection point in exactly one node.

![Figure 3: A 2 order Fano plane with 7 nodes\textsuperscript{[45]}](image)

Figure 3 is a pictorial representation of the FPP Quorum with a Fano Plane shape, which is an
order 2 projective plane consisting a set of 7 nodes (i.e., \(n=7\)).

In Grid Quorum, assuming \(|S| = k^2\), for some \(k\) and nodes placed in a square matrix, where
both the value of \(k\) and nodes are integer. In that scenario, a Quorum system over \(S\) is
constructed with having the following characteristics:

- \(S\) consists of a set of subsets with the form \(Q_{i,j}\)
- each \(Q_{i,j}\) contains all elements in row column namely \(i\) and \(j\) respectively, where \(1 \leq i \leq k\)

and \(j \leq k\)
- every quorum system contains \( n \) number or quorums, where \( n=k^2 \), with the size of \( 2k-1=O(\sqrt{n}) \), and
- every quorum has an intersection point with every other quorum in 2 minimum of nodes.

A related point to consider is, to minimize the size of quorum intersection, a slightly different quorum system can be constructed containing \( k = \sqrt{n} \) quorums \( Q_i \) (\( 1 \leq i \leq k \)) such that \( Q_i \) contains all nodes from row \( i \) and exactly one node from each row \( j > i \) \([45]\).

Another quorum system, B-Grid \([47]\), assumes grid on a rectangular box of \( R \) rows and \( C \) columns, such that rows are grouped into \( b \) bands of \( r \) rows \( (R = br) \), where band \( j \) (\( 1 \leq j \leq b \)) contains rows \( (j-1)r+1...jr \) \([45]\). With that being denoted, a mini-column \( (c, j) \) can also be defined that is mainly the intersecting point of column \( C_c \) and band \( j \).

Then, the quorum system containing \( b \) quorums \( Q_j \) (\( 1 \leq j \leq b \)), each containing one mini column from each band and one node from each column in band \( j \) \([45]\). As a consequence, it is easily observable that in B-Grid, every quorum has exactly \( br + C_c - 1 \) nodes.

**Figure 4: B-Grid Quorum system \([45]\)**

Figure 4 is a B-Grid pattern containing a set of 120 nodes (i.e., \( n=brc=120 \)), with column \( C_c = 15 \), band \( b=4 \) and rows \( r=2 \) per band.

**2.2.3 ACID Property and the Alternatives of ACID**

ACID (Atomicity, Consistency, Isolation, and Durability) is a set of properties that guarantee that database transactions are processed reliably \([19, 21]\). ACID features are usually implemented in relational databases such as MySQL and Microsoft SQL server products. In relational databases, data is stored in tables, consist of rows and columns, which have their own data types and they are precisely defined. The tables can have relationship with other tables, and structured query language (SQL) is used to retrieve data. On the other hand, NoSQL handles unstructured and multiple data types. A single node is no more responsible for dealing with all data transaction. Therefore, ACID cannot be enforced on NoSQL databases.

The article “BASE: An ACID Alternative” \([11]\) has made few important arguments on how BASE can adopt itself in distributed environment as an alternative way to maintain
consistency. The author mentioned, “BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is well manageable and leads to levels of scalability that cannot be obtained with ACID”.

Figure 5 illustrates the sample schema of consistency consideration for BASE. In the figure, table A contains user information including the total sold and bought amount. They are the running total amounts. Table B contains every transaction that were made, relating the seller, buyer and the amount of the transaction. According to the author’s explanation about the schema, “these are gross oversimplifications of real tables but contain the necessary elements for illustrating several aspects of consistency. In general, consistency across functional groups is easier to relax than within functional groups”\cite{11}. The illustrated schema is made up of two functional groups: users and transactions. Once an item is sold, a row is added to the table B and the number of buyers and sellers are increased. The total bought and sold columns in the table A can be considered a cache of the table B. It is used to enhance system’s performance. In this given scenario, the constraint on consistency could be relaxed and the buyer and seller expectations can be set so their running balances do not reflect the result of a transaction immediately\cite{11}. From this example, the fact is established that BASE failed to provide strong consistency.

Figure 5. Schema of consistency considerations for BASE\cite{11}

In the article\cite{10}, written by Deka Gnesh Chandra, an analytical experiment on BASE feature of some NoSQL databases such as BigTable (used by Google app engine) and HBase (which runs over Hadoop) has been conducted. The author used ‘The Yahoo! Cloud Serving Benchmark (YCSB)’ platform for benchmarking of NoSQL. His research work shows the measured performance, scalability, flexibility, complexity and functionality of some NoSQL. He also has demonstrated the level of consistency of few leading NoSQL databases by analysing their consistency. The author was able to make some important observation of some NoSQL (i.e., the downside of NoSQL), as well as recommendations based on observations for selection of a NoSQL for specific purposes. According to the data I gathered
and findings from various sources, the following observation is made about the databases using BASE [10, 29, 33]:

- BASE shows scaling at about half of linear capacity throughout.
- BASE scales poor for the unbalanced workload. According to the study, “for heavy-read workloads (95% read, 5% write) Column Family databases shows excellent writing abilities, but its reading performance is poor, since this product was optimized for writing resulting to lots of concurrent I/O when reading”.
- Databases with BASE use large amounts of memory and it has to perform lots of disk I/O in read heavy workloads, thus, leading to a highly decreased performance.
- Reads in BASE are slow. It has tuneable consistency and sacrifice consistency in favour of availability and partition tolerance. After all it is never able to maintain strong consistency, always providing eventual consistency; ‘stale’ data is ok.

However, as discussed earlier in this report, another popular consistency maintaining technique used in many large-scaled distributed systems is Quorum. Distributed quorums are popular among eventually consistent storage systems in data centres. The article “Comparison of consistency approaches for cloud databases” [9] mentions that quorum replication allows storage systems write a data item by sending it to a set of replicas and read from a possibly different set of replicas. They are called “write quorums” and “read quorums” respectively. The authors have studied the performance of Quorum with respect to some parameters and demonstrated the weaknesses and advantages of the technique. The summarized points from their results of the experiments are presented below:

- Quorum technique uses a distributed point of entry for write request
- Any node in a network can act as a primary node with the secondary nodes and servers are allowed to participate in all read-write requests.
- Using 70-30 read-write ratio and six nodes in the experiment, a server following Quorum-based approach spends 34 ms per operation, on average.
- Quorum servers usually spends longer amount of times than others.
- Even though the Quorum approach is specifically designed to share the load of the system, it still shows worst performance. It happens because every quorum server is requested to participate in every read and write operation. “At a higher request arrival rate, all quorum servers are busier than the servers for the other approaches”, according to the authors.
- The Quorum approach neglects the effect of higher write request, as response time varies from 204 ms for a 90-10 read-write ratio to 265 ms for a 50-50 read-write ratio.

After synthesising the experiments and observations in the research works, the following statements are inferred:

Although Quorum, in general, provides eventual consistency, in some certain situations this approach is also able to provide fully consistent state in database. The approach shows better performance if the percentage of write request in subsequent read or write requests is high [9, 34].


2.3 YCSB: Database Benchmarking Tool

2.3.1 Description

With the growing number of new databases including BigTable, Azure, Cassandra and many more, it is difficult to decide which database is proper for any distributed application, partially because the features differ among databases, and there is no divine way to measure the consistency level of one database versus another.[35]. This necessity convinced the researchers to develop YCSB which includes a common set of workloads to examine the behaviour and performance of different cloud database systems.

YCSB is a standard benchmarking framework to assist in the evaluation of different cloud systems and NoSQL databases.[32]. The YCSB project has been developed by a group of computer scientist in the research division of Yahoo Inc. It was written in Java and includes features where users can implement the existing API to benchmark their own database. YCSB does not provide any GUI to interact with its users. The latest release (version 0.10.0) of the framework resides in Github repository that allows users to clone or download the source file to desktop. The YCSB framework is comprised by two areas.[35];

The client, an extensible workload generator and the core workloads which is a set of workload scenarios to be executed by the generator.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Update heavy</td>
<td>Read: 50%, Update: 50%</td>
</tr>
<tr>
<td>B-Read heavy</td>
<td>Read: 95%, Update: 5%</td>
</tr>
<tr>
<td>C-Read Only</td>
<td>Read: 100%</td>
</tr>
<tr>
<td>D-Read Latest</td>
<td>Read: 95%, Insert: 5%</td>
</tr>
<tr>
<td>E-Short ranges</td>
<td>Scan: 95%, Insert: 5%</td>
</tr>
</tbody>
</table>

Table 1 presents all the six workloads type included in the core workload. Most of the cases, the core workloads are enough to judge a system performance and consistency level. For this thesis, three different types of workloads were selected- Workload A, Workload B, Workload D, and each of them having different consistency parameter. Workload C was excluded from the selection, as it’s read operation is already included with other workloads. Currently, scan transactions are not directly supported by Riak, as there is no suitable mean to perform them properly.[30]. This will not cause the benchmark to fail, it simply will not perform any scan operation at all; these will immediately return with a Status.NOT_IMPLEMENTED.[30]. However, this implementation can be done with additional configuration which involves enabling Riak Search on every node within the
with this reason in mind, and also to avoid excessive technical complication in this regard, I have not performed *scan* operation in any of the database, which means *Workload E* was also removed from the choice. After all, all workloads were selected such a way that it suits with a typical today’s modern applications.

However, the core workload does not cover all the system aspects and database scenarios, and that is why extending the client becomes necessary when I have to define different workloads. This feature enables developers to test any NoSQL database and get the benchmarking output. At this moment, YCSB supports 19 different NoSQL databases including Cassandra and Riak. In order to conduct any experiment with additional database, several manipulations, for example, creating project module, implementing java methods and database query are necessary in the core project. However, no further database interface layer is considered to be implemented since the databases to be tested are already packed with the YCSB.

**Figure 6. YCSB client architecture**

Figure 6 illustrates the architecture of YCSB client. The client framework does two main jobs- generating data that needs to be loaded into database and generating the operation which eventually form workload.

Workload executor runs multiple client threads. Each thread executes a sequential series of operations by making calls to the database interface layer, both to load the database and to execute the workload. Besides, each client thread measures the latency and thereby achieving the throughput of the operations. A series of properties such as name/value pairs in client define its operation. By convention these operations are defined into two properties.

- **Workload properties**, is used to define the workload. For example, the read-write mix, distribution of use, and the number and size of the field of database.
- **Runtime properties** define the specification of the properties to a given experiment. For example, the number of threads and which database to use etc.

Furthermore, to run the workload against each database, the following procedures were followed. This common set of procedures are applicable for both Cassandra and Riak database.
2.3.2 Setting up the Database System

An existing database with table is required in the beginning of the process. I gave the table name `usertable`, as the core workload assumes that there is a table with this name. The interface layer of the particular database is responsible for reading and inserting data in that table. To complete this operation, the interface layer needs to know the underlying structure of the storage. For example, in Cassandra, I had to define `column families` besides `keyspaces`. Thus, I created a column family and give the family a name. This is how the database access layer knows to refer to the `values` column family, either because the string `values` is passed in as a property, or because it is hardcoded in the database interface layer [30]. Cassandra also allows to create multiple replicas and clusters to distribute record across them- which I needed to define before the loading phase. Similarly, Riak provides simple query to define the number of nodes and clusters in which the records will be allocated. To get interesting output, I executed the operation under different workloads for both data stores. This is how the consistency level varied from one operation to another as well as one data store to another.

2.3.3 Choosing the Database Interface Layer and Workloads

The YCSB Client provides a simple dummy interface layer, `com.yahoo.ycsb.BasicDB`, which is a java class. It executes read, insert, update, delete and scan calls generated by the YCSB Client into calls against the database's API [30]. I specified the class name on the command line, and YCSB client loaded that class. Any properties or parameters specified on the command line can be passed to the interface instance. For example, to configure the interface layer, I passed the hostname of the database I benchmarked. However, specifying the appropriate workload were necessary in order for loading the record into the database. The workloads are designed to handle two phases- `loading phase`, in which the data will be inserted into the database and `transaction phase`, in which the operation (e.g., read and update) will be executed against the inserted record.

2.3.4 Loading the Data and Executing the Workloads

To load any amount of record into the table, the workload type must be decided. Depending on the workload, the throughput (ops/sec) and latencies vary. By default, the YCSB client inserts 1000 records. However, with the appropriate command argument it is possible to define a custom number of record to be inserted (i.e., `recordcount=100000`). Loading phase will start inserting record once it receives the command specifying the local host and port number in which the database cluster is running. The command given below on windows shell loads data into the Cassandra database.

```
"C:\Python27\python.exe" \bin\ycsb load cassandra2-cql -P workloads/workloada -p hosts=127.0.0.1 -p port=9042
```

A successful loading phase shows the following output.
The figure displays the latency and throughput of workload $A$ during its loading stage. Additional information such as cluster and datacentre are also noticeable from the output.

The final stage (i.e., transaction phase) of the benchmark is to run the workload. The command to execute the workload is slightly different from the aforementioned one:

"C:\Python27\python.exe" bin\ycsb run cassandra2-cql -P workloads/workloada -p hosts=127.0.0.1 -p port=9042
3 Results

3.1 Data Inconsistency

The following sub sections contain the reasons and motivation for data inconsistency. By searching and evaluating the available scholarly articles in the chosen topic area, the following factors are found that cause data inconsistency.

3.1.1 Data Redundancy

Data redundancy is a condition created within a database or data storage technology in which the same piece of data is held in two separate places [23]. Network traffic manifests large amount of redundancy when many users on the Internet access similar content [36]. Although data redundancy is key to preventing data loss and achieving fault-tolerance in cloud storage [37], many distributed systems (operating on application and object level) are working to explore how to reduce the chances of this redundant content from network links and improve network efficiency. For example, web proxy caches and the more recent P2P caches store frequently accessed objects and serve repeated requests from cache [36].

However, with the given definition of data redundancy, there are different classifications based on what is considered appropriate in database management, and what is considered excessive or wasteful [23]. Wasteful data redundancy may occur when a given piece of data is not meant to be repeated, but eventually becomes duplicate due to inept coding and, or process complexity [23]. Sometimes developers do not consider it acceptable for data to be stored multiple places. If so, there is no central and master field of space for the data. That means, there is no way to update data all of the places and the data isn’t redundant through one central access point. This situation leads to big problems with data inconsistency, where one update does not automatically update another field. As a result, pieces of data that are supposed to be identical end up having different values [23].

According to the gathered information and supporting arguments found in various sources, it is convincing enough to say that data inconsistency is likely to occur if there is data redundancy. Data redundancy occurs when the database file has redundant and unnecessarily duplicated data. That’s why one major goal of good database design is to eliminate data redundancy [2].

3.1.2 Network Latency

Managing geographically dispersed deployments of complex multitier applications involves dealing with the substantial effects of network latency [26].

Network latency is the delay occurring in data communication over networks. Network connection suffering from small delays are called low-latency network and long delays are high-latency. High latency greatly affects the communication bandwidth in network pipe. It is also highly responsible for creating bottlenecks in network communication as well as preventing the data from taking full advantage of the network pipe that effectively loss or decrease the communication bandwidth. The impact of latency on network bandwidth can be temporary or persistent based on the source of the delays [24]. Whether it is high latency or
Low latency, delays adversely impact the performance of the response time of the data being sent from one node to another. Moreover, a high level of latency can cause loss of connectivity to one or more partitions [25]. “If a network failure renders one or more of the data stores inaccessible during a transaction, an application updating data in a system that implements strong consistency may be blocked until every data store becomes accessible again [25].” So, it is obvious that high latency makes it difficult to deliver data packet in consistent manner [27].

Latency can impact the response of end-to-end multitier applications [26]. An example is considered, based on the results presented in the article [26], where a traditional multitier application implements a JavaBeans-based online auction site RUBiS [28] showing in Figure 8.

As can be seen from the figure, RUBiS has three tiers: a front-end Apache-based Web server, a middle Tomcat-based application, and a back-end MySQL-based database. Across the MySQL database tier, the application logic is distributed.

However, having the RUBiS application with the following deployments architecture-web server is in San Diego, CA, the back-end database server in New York City, and the application server co-located with either the front-end or the back-end servers, Table 2 shows the response times of a few RUBiS transactions for both deployments [26].
Table 2. Average response time for RUBiS application [26]

<table>
<thead>
<tr>
<th>Transaction</th>
<th>San Diego (sec)</th>
<th>New York (sec)</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>SearchItembyReg</td>
<td>5.70</td>
<td>0.16</td>
<td>-97%</td>
</tr>
<tr>
<td>ViewBidHistory</td>
<td>2.31</td>
<td>0.19</td>
<td>-92%</td>
</tr>
<tr>
<td>ViewUserInfo</td>
<td>4.13</td>
<td>0.20</td>
<td>-95%</td>
</tr>
</tbody>
</table>

According to the table 2 data, it is obvious that even with such a simple application, the impact of network latency changes on end-to-end response time is dramatic and not trivial to predict without a detailed understanding of the application [26]. The significant changes in response time proves that data were not updated both location synchronously, and therefore, users see the stale data.

3.2 Satisfactory Level of the Approaches

For the purpose to gain the satisfactory level of two approaches, following configuration settings are used for each of the workloads:

- Total records: 100,000,000
- The total size of a record: 1 Kb
- Cluster: 4-nodes cluster
- 10 fields of 100 bytes each per record

According to the conducted test and consequent output of databases with different workloads, the collected data is presented below with line graphs.
3.2.1 Throughput by Workload: Test 1

![Figure 9: Throughput by Workload A](image)

Figure 9 demonstrates the latency versus throughput curves for each read and update operation for both Cassandra and Riak. The figure illustrates that for both data stores, operation latency increases as throughput increases. For the read operation, Cassandra achieved lowest latency than Riak, although the increasing rate is very slow for both systems per throughput. In Cassandra, the maximum number of throughput (i.e., 2000 ops/sec) differs only 4 ms from the initial throughput (i.e., 500 ops/sec), whereas for Riak it is 5 ms.

For update operation, the system’s performance differs excessively from one another. Cassandra shows interesting increasing of latency for every 1000 ops/sec. The average latency is almost similar for both minimum and maximum numbers of throughput. On the contrary, Riak exposes the dramatic difference from Cassandra. It’s latency boost rate is double than Cassandra. To gain the highest throughput, Cassandra requires only 2 ms, whereas Riak stands in need for 13 ms.

According to the observation from the both scenarios, Cassandra achieves highest number of throughput for least latency for each operation.
3.2.2 Throughput by Workload: Test 2

Figure 10 shows the output of workload B. For each operation, Cassandra shows lower read latency at high throughputs. However, in this test, operation latency didn’t increase as throughput increase, in both systems. Cassandra behaved adversely to Riak, which demonstrated that- for Cassandra, latency drops to the lowest for highest throughput and for Riak, latency boosts up as throughput becomes maximum. The read latency is relatively higher for both data store. For update operation, latency is surprisingly low in Cassandra. Once the throughput reached 1000 ops/sec, the latency falls down on 1 ms and remained the same rate as throughput continued incrementing by 500 ops/sec. In Riak, both operations maintained consistent manner while increasing latency according to the throughput.
3.2.3 Throughput by Workload: Test 3

![Throughput by Workload D](image)

Figure 11: Throughput by Workload D

Figure 11 presents output for insert and read operations. In this final test, Cassandra performed surprisingly poor while inserting record. More than 30 ms was required to insert data for each of the throughput amount. However, for read operation, Cassandra had outstanding achievement. No latency was calculated until it started dealing with 2000 ops/sec throughput, and this rising was not sudden change, since 1 ms is the minimum latency considered. Then, Riak brought similar output to its insert operation. So it is obvious from the Cassandra’s outcome that its read and insert operation varies from each other to a great extent.

3.2.4 Satisfactory Level of Consistency

According to the test results, Cassandra required much less time than Riak for reading, inserting and updating value from one node to another. Figure 9, 10 and 11 show that Cassandra had better performance than Riak while mapping the latency versus throughput with different workloads. The low latency behaviour of Cassandra tells us that it has better consistency maintenance ability- because, this way the data store has least chance to appear with conflicting versions of the same value in different places. To show the satisfactory level of both databases in a distinct manner, I generated a standard level of scale having 4 levels: 1 to 4, where the higher the level, the better the performance. Furthermore, I defined the levels by dividing the latency in every 10 ms (e.g., latency 0 - 10 falls into the level 4: the best, and 31- 40 falls into level 1: the worst). The table below summarizes the test results.
In the beginning of the report, I have addressed two consistency approaches (i.e., BASE and Quorum) that Cassandra and Riak follow to maintain their consistency. On Table 3, Cassandra had the highest satisfactory levels compared to Riak. We, therefore, draw a tenable implication that the Quorum approach for Riak leaves an option to optimize its performance in terms of consistency.

### 3.3 Optimization of Quorum

#### 3.3.1 Test Environment

The aim of these tests are to get Quorum performance in different use cases. The use cases are variations of workloads, threads and target operations per second. These tests help me to realize Quorum’s effectiveness in different input scenarios and decide whether Quorum approach is effective or not for those particular use cases. It is expected for me to come into a level of knowledge where suggestions regarding Quorum’s efficiency towards better consistency can be provided. The experimentation environment setup for these tests follows the similar arrangement as described in section 2.3.2 to 2.3.4, with the following additional configuration:
- Number of threads: 5, 10, and 15.
- Number of Target (ops/sec): 500, and 1000.

---

**Table 3. Satisfactory level of Cassandra and Riak**

<table>
<thead>
<tr>
<th>Workload</th>
<th>Operation</th>
<th>Cassandra</th>
<th>Riak</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Read</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Update</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>Read</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Update</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>Insert</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
3.3.2 Result of Experimentation: Test 1

The average latencies recorded for 4 different read-update ratios are shown in the table 4. The number of thread is 5.

Table 4. Average Latencies for 5 threads

<table>
<thead>
<tr>
<th>Read-Update Ratio (%)</th>
<th>Average Latency (ms) with Target 500 (ops/sec)</th>
<th>Average Latency (ms) with Target 1000 (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50-50</td>
<td>18</td>
<td>29</td>
</tr>
<tr>
<td>70-30</td>
<td>14</td>
<td>22</td>
</tr>
<tr>
<td>80-20</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>95-5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

The size of read and update operations in the workloads has a significant impact on the latencies. The average latency decreases from update heavy (50-50) workload to read-heavy (95-5) workload, both for target 500 and 1000. This trend continues as straight: more update – more latency and more target – more latency.

3.3.3 Result of Experimentation: Test 2

The average latencies while having 10 threads, are shown in table 5.

Table 5. Average Latencies for 10 threads

<table>
<thead>
<tr>
<th>Read-Update Ratio (%)</th>
<th>Average Latency (ms) with Target 500 (ops/sec)</th>
<th>Average Latency (ms) with Target 1000 (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50-50</td>
<td>27</td>
<td>34</td>
</tr>
<tr>
<td>70-30</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>80-20</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>95-5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

According to the table, there is no significant change in the trend while changing the number of threads from 5 to 10. The average latency increases while there is more update than read
operations. For the read heavy workload, there is no big difference between latencies, showing better performance for read heavy workloads.

3.3.4 Result of Experimentation: Test 3

Table 6 below shows the average latencies in operations per second for 15 threads.

<table>
<thead>
<tr>
<th>Read-Update Ratio (%)</th>
<th>Average Latency (ms) with Target 500 (ops/sec)</th>
<th>Average Latency (ms) with Target 1000 (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50-50</td>
<td>28</td>
<td>35</td>
</tr>
<tr>
<td>70-30</td>
<td>26</td>
<td>28</td>
</tr>
<tr>
<td>80-20</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>95-5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

The latencies do not follow the same trend as did for the previous two tests. The average latencies for thread 10 and 15 are almost similar, with very little changes. After a certain level, the number of threads has no significant impact on latency, but the number of operations per second.

3.3.5 Experiment Evaluation

From the test results above, it is clearly visible that the latency is higher when the update got higher ratio in the workloads. The quorum approach in this case, is appeared to be inconsistent for update-heavy operations. A good example of this use case can be the user session, where recent sessions are recorded in session store. Most of the web application required this user session state in their implementation. On the other hand, some applications are not heavily dependent on update; instead reading data fast and consistently get more importance. Example of this type of application is tagging photo, where most operations are to read tags. The quorum approach got a better performance for this kind of applications. As the results above, quorum performed well with least latency for read-heavy and update-light workloads.

3.3.6 Suggestion for Better Performance

Latency is the cost to pay for consistency. For $N$ number of replicas, a client either could read one replica only and does not require any checking for consistency or could wait for a Quorum read, that requires the system to confirm about the consistency across multiple replicas. The latter option takes time, that is the latency I can see from our experiments.
From the experiments, above, the update-heavy workloads costs more in the form of network latency. Varieties of read-update ratios have been chosen to find the correct tune for latency efficiency application of Quorum. Since better performance has been achieved through less update workloads, application with read-heavy behaviour are the best choice for applications that run on database based on the Quorum. Results from these tests convinced to conclude our findings, which determine the Quorum's inability to deal with the inconsistency problem in NoSQL databases.
4 Discussion

In this section, I briefly explain the implications of the results. I interpret the findings, answer the questions stated in the introduction, and present arguments to support our findings. Additionally, I describe how the research topic relates the answers and fit with current knowledge, as well as a suggestion for future work.

4.1 Data Inconsistency Factors

The existing research works concerning inconsistency issues in NoSQL databases contains some common key causes that makes the distributed application inconsistence. Papers relating this issues have extensive amount of experiment that derives the fact that causes the data inconsistence. Data redundancy, as one of the mostly found reason that usually occurs in geographically dispersed data centres. Because of the different types of users (e.g., multi user, work group, and enterprise) interact with such diverse and unstructured data which eventually make the transaction process complex, thus developers fail to practice efficient database design. The defective operation by the application and database designer cause the replica remaining with the redundant data.

Network delay, obviously is an ordinary drawback that prevents data from being updated in time. From the benchmarking of Cassandra and Riak databases, this nature is observed. Although the latency from node to node varies, the usual characteristics they carry is, there is always few milliseconds time required to send the update from one storage location to another. The similar observation is noted when the average response time was measured in a multi-tier deployment environment. The latencies imply with different use cases (e.g., searching item, and viewing user information), which is an impact on updating data in synchronizing manner.

I have restrained the research for the first research problem within the area of distributed application. I rely on the experimentation results presented by researchers, to identify the factors for data inconsistencies. The presented key factors are relevant to the distributed application, and NoSQL databases. As discussed in the section 3.1, the developer’s issues for causing data redundancy, and high network latency both are noticed in non-relational databases and application in distributed environment. So the explored reasons for data inconsistency fits within the current research topic.

4.2 Satisfactory Level of the Approaches

There are two consistency maintaining approaches (i.e., BASE and Quorum) have been introduced earlier in this paper. The benchmarking tool YCSB requires pre-existing table created in the NoSQL databases in which it will load and execute the workloads. The final output after executing the workloads shows the latency in milliseconds for different throughput size. For all three types of operation (i.e., read, update and insert), and defined node-cluster size, Cassandra and Riak shows different network latencies. In Riak, both read and update operations require few milliseconds (varying from 4 to 30 ms) to dispatch from one cluster to another, whereas Cassandra demands incredibly less amount of time to read
and update the data. However, an exception is seen in test 3, where insert operation in Cassandra takes much longer time to be performed than Riak.

The satisfactory level of consistency is realized through the latency versus throughput in each database. Operations that takes longer average latency is considered as less consistent, thus the satisfactory level of consistency is lower.

According to the findings in section 3.1.2, one of the recognized factors for data inconsistency in NoSQL databases is network latency, or in another words, time delay to reach (or update) data from one data location to another. This behaviour fails to deliver the latest data to the users, in which case ensuring strong consistency becomes a matter of issue. So the network latency is a feasible choice for judging the satisfactory level of consistency, and according to the conducted test, Quorum approach is entitled with the poor consistency maintenance protocol than the other.

However, benchmarking each of the database provides a decent answer to the problem with consistency quality of BASE and Quorum, and also guide me to remark their satisfactory levels.

4.3 Suggestion of Quorum Optimization

After analysing the satisfactory level of the consistency, the decision is settled that Quorum leaves a room for further optimization. An important point to consider is that for any consistency approach, latencies varies from one use case to another (e.g., read heavy, write heavy), so Quorum does. Testing the Riak database with various read-update ratio shows us variation in latencies. The common observation from the test is for lighter update ratio, latencies decreases. Because average time per operations is considered, whenever the operations of update ratio are low compared to read operations, the system spends less amount of time on average per operation. So with least update request, the system achieves better consistency.

An important point to note that consistency maintenance approach like Quorum is an underlying design tied with databases, which is practically not authoritative for everyone to change or alter. However, the benchmarking tool already provides workloads with various nature that allows us to test any system with different read-update ratio. Having that feature in workloads and experimenting them with Riak database, can give a suggestion on how to implement the system to improve the consistency of Quorum depending on use cases.
5 Conclusion

In this paper, some factors that causes data inconsistency in distributed application are explored. The consistency performance of two approaches (i.e., BASE and Quorum) are analysed and reported their satisfactory level of consistency. Analysing the satisfactory level of consistency has established the fact that Quorum approach has room for improvement suggestion. Afterwards, a suggestion was proposed saying how the Quorum approach can achieve better performance.

Identifying the consistency level is a valuable achievement for this research, as developers can have better plan and action while designing cloud application and database. The optimization suggestion that has been proposed can also be contemplated with distributed application’s design principle.
6 Future Works

6.1 Finding more Inconsistency Reasons

Although the existing literatures has mentioned some reasons for data inconsistency in distributed application, there are still many possibilities to detect more reasons behind this issue. For this research the experimentation already done by researchers are considered. To improve the version of this part of the work, some more experimentations can be done individually. Some suggestions on future experimentation regarding this research issue can be as follows.

1) Making observation with more use cases in addition to the scenarios (i.e., Transaction column) mentioned in table 2. For instance, user creating data, updating data, delete data etc.

2) Adding more geographical locations to deploy the application to see more interesting impact on network latency, as this may affect the change on end-to-end response time.

3) Moreover, the experimentation can be conducted with different type of auction site (e.g., eBay, eBid) with different configurations than the presented figure 8.

6.2 Measure Consistency Level with Other NoSQL Databases

Satisfactory level of consistency for this research was realized through the benchmarking of Cassandra and Riak. However, the test can be expanded into other NoSQL databases such as Amazon Dynamo and Azure DocumentDB. Furthermore, new workload can be defined to replace Core Workload by extending the Workload class of the YCSB. This way the satisfactory level can be more plausible, as the additional workloads will allow to understand the performance trade-offs of different systems.
7 Bibliography


