

**Bachelor's thesis**

IU International University of Applied Sciences

Computer Science

Comparative analysis of Firebase Realtime database vs  
Firestore for mobile chat applications.

Mohammad El Fayez

Enrolment number: 92003608

Jabal Arafat street, 94, Amman, Jordan

16197, Um al Amad

Supervisor: Prof. Dr. Markus C. Hemmer

Date of submission: 20-03-2024

## Abstract

Executive summary using the following keywords:

**Purpose.** This study aims to compare the scalability, real-time capabilities, offline capabilities, querying efficiency, and data security of Firestore and Firebase Realtime Database, to determine which is more suited for supporting mobile chat applications.

**Value.** The findings provide insights for developers and decision-makers in the mobile app development industry and chat app development industry by highlighting the trade-offs between Firestore and Firebase Realtime Database across essential performance and functionality metrics.

**Methods.** The research utilizes empirical and practical testing, by developing two applications, one utilizing Firestore, and the other utilizing Firebase Realtime Database and testing specific aspects of each research question on the application.

**Key findings.** Tradeoff between scalability and real time updates

**Conclusion.** The investigation revealed that Firebase Realtime Firestore demonstrates superior scalability, data security, and offline capabilities while Firebase Realtime demonstrates superior querying and real-time updates.

**Keywords:** Firebase, Firebase Realtime, Firestore



# Table of Contents

<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vi</b>
<b>List of Equations</b>	<b>vi</b>
<b>List of Abbreviations</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research Objective	2
1.2 Value	3
1.3 Target audience	4
1.4 Scope	4
1.5 Constraints	4
1.6 Structure of the Document	5
<b>2 Literature Review</b>	<b>6</b>
2.1 Theoretical Background	6
2.1.1 Firebase	6
2.1.2 Cloud Firestore and Firebase Realtime Database.	6
Cloud Firestore Data Model	7
Firebase Realtime Database Data Model	7
<b>3 Research Design</b>	<b>15</b>
3.1 Testing metrics	16
3.1.1 Scalability and Real-time updates testing metrics.	16
3.1.2 Querying metrics	16
3.2 Requirements Engineering	17
3.3 Requirements Specification	17
3.4 Technology	18
3.5 Development and testing	19
3.5.1 Database models and configuration	19
3.5.2 Query Testing Process	20
3.5.3 Scalability Testing	22

<b>4</b>	<b>Results &amp; Discussion</b>	<b>25</b>
4.1	Query Testing Primary and Derived Data	25
4.2	Query Discussion	27
4.3	Scalability Primary and Derived Data	29
4.4	Discussion for Scalability	29
4.5	Synopsis	30
<b>5</b>	<b>Conclusion</b>	<b>32</b>
5.1	Critical Reflection	32
5.2	Recommendations for Future Research	32
<b>6</b>	<b>References</b>	<b>33</b>

## List of Figures

Figure 1. Logical sequence of chapters	5
Figure 2. Firestore document store model for chat applications.	7

## List of Tables

Table 1. Requirements for Chat Application and testing	17
Table 2. Execution time for CREATE queries.	25
Table 3. Execution time for READ queries.	25
Table 4. Execution time for UPDATE queries.	26
Table 5. Execution time for DELETE queries.	26
Table 6. Execution time for filtering by sender and timestamp.	27
Table 7. Read response time.	29
Table 8. Write response time.	29

## List of Equations

## List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application program Interface
BaaS	Backend-as-a-Service
CAP	Consistency, Availability, Partition tolerance.
CRUD	Create, Read, Update, Delete
ISO	International Organization for Standardization
SOC	System and organization controls
JSON	JavaScript Object Notation
Mib	Mebibyte
PII	Personally Identifiable Information



# 1 Introduction

The rise of mobile chat applications has revolutionized how we communicate with each other, allowing users to engage in real-time instant conversations, group chats, and multimedia sharing. The importance of those applications cannot be understated, as they play a pivotal role across many platforms, such as social media, gaming, and business collaboration. A study by Dixon showed that as of January 2023, WhatsApp alone had two billion monthly users,<sup>1</sup> which further illustrates the popularity of chat applications.

Despite the wide range of mobile chat applications available, they all share the same core functionalities that are vital for any type of mobile chat application, such as real-time messaging and user authentication. Those functionalities serve as a foundation for generating the key requirements the chosen database must meet to ensure that the application functions properly. The key requirements include:

- Robust scalability
- Efficient real-time capabilities
- Efficient querying capabilities
- Effective Security measures and mechanisms

Those requirements will form the cornerstone of the research and will act as the key criteria for the comparative analysis.

## **The relevance of Firestore and Firebase Realtime database**

Backend as a service solutions became increasingly popular in the development realm as they offer an easy, time-efficient approach to developing the back-end infrastructure of an application, eliminating the need for manually creating and managing the servers and databases. BaaS platforms allow developers to focus on front-end features by providing a ready-made backend that can be accessed through APIs.<sup>2</sup>

Firebase provides many tools that can be used for various aspects of web and mobile application development. It enhances traditional API capabilities by adding features like messaging and authentication, which reduces both cost and development time.<sup>3</sup> Firebase's ease of use and flexibility make it suitable for all types of chat applications. Firebase offers two types of databases, Firestore, and Firebase Realtime

---

<sup>1</sup> (Dixon, 2023)

<sup>2</sup> (Dudjak & Martinović, 2020, p. 2)

<sup>3</sup> (Tram, 2019, p. 20)



database, despite having many functional and architectural differences, both databases meet the key requirements that were defined earlier, which makes them natural choices for developers looking to develop responsive, high-quality mobile chat applications.

However, there is a gap in comprehensive literature and analyses that directly compare both databases in general, let alone in the context of mobile chat applications, and since Firestore and Firebase Realtime database operate differently and have their own advantages that can be suitable for certain types of chat applications, this gap presents a problem for developers as they might have to rely on general assumptions that do not fully account for the unique demands of their mobile chat applications. The gap presents an opportunity for a detailed comparison that analyses the strengths, weaknesses, and use cases for both databases.

## **1.1 Research Objective**

The main objective is to conduct a comparative analysis between Firestore and Firebase Realtime database. This analysis aims to explore and compare how both databases perform across various core aspects that are essential for mobile chat applications. The research questions below are structured to directly explore and compare each aspect, providing a detailed understanding of the capabilities of each database.

### **1. Which Database scales more effectively as the number of users and messages increases?**

As the number of messages and concurrent users increases, it becomes vital for a database to handle the growth without compromising the performance of the application, investigating the scalability of both Firestore and Firebase Realtime database will help determine which is more suitable for applications with expected high traffic.

### **2. Which database demonstrates more efficient real time capabilities for mobile chat applications?**

A database's ability to reflect real time changes quickly, such as message and profile updates is vital for a chat application. Equally important is its ability to efficiently resolve conflicts that occur when multiple users try to modify or delete the same data at the same time. Assessing the crucial factors will help determine whether Firestore or Firebase Realtime database is more suitable for applications that require instant and reliable communication among users.

### **3. Which database offers better offline capabilities for mobile chat applications?**

A database's ability to handle scenarios where an offline user tries to send, modify, or update data and then immediately synchronize these changes and resolve

any conflicts after the user regains connection is vital for a chat application. This question seeks to assess and determine whether Firestore or Firebase Realtime database handles those situations more efficiently.

**4. Which database offers more efficient querying capabilities for mobile chat applications?**

Chat applications involve a continuous stream of messages between users, requiring fast and efficient querying to fetch and write new messages as well as perform compound queries for features that need queries that combine two or more fields. Comparing the simple and compound querying capabilities of Firestore and Firebase Realtime database helps determine which database is more suitable for applications with large datasets.

**5. Which database provides more effective security measures and mechanisms to protect user data?**

Data security is an important aspect in all types of applications, this question investigates and compares Firestore and Firebase Realtime database in terms of their security as well as compliance with data protection regulations.

The findings from the questions will not necessarily provide a best choice for all chat applications; instead, it can be used to help developers and companies choose the most suitable database that aligns with the requirements of their mobile chat application.

## **1.2 Value**

The value of this thesis is determined by the following criteria:

- 1. Practical application:** This thesis will provide valuable insights into the strengths and weaknesses of both databases, as well as recommendations to developers and companies that can aid them in choosing the right database, while this research is specifically tailored for chat applications, the insights can also be beneficial for a broader range of mobile applications thus expanding the scope and relevance of this thesis.
- 2. Optimization:** The findings will contribute to the overall quality and performance of mobile chat applications as the developers will be better informed about choosing the most suitable database for their specific needs.
- 3. Academic contribution:** Due to the lack of research directly comparing both databases, this thesis greatly contributes to the field of software engineering, especially in the realm of database technology for mobile applications.

### 1.3 Target audience

The identified audience for this thesis encompasses:

1. **Mobile application developers:** The primary audience, as they will benefit the most from this thesis.
2. **Students:** Students studying mobile application development or database technologies can greatly benefit from this thesis.
3. **Database technology companies:** Companies like Google, who own Firebase, can use this thesis to understand how their products work in real-world scenarios and make improvements.

### 1.4 Scope

The scope of this thesis involves the following areas:

1. **Industry and application area:** The thesis primarily focuses on the mobile application industry, with a specific focus on Firestore and Firebase Realtime database for mobile chat applications.
2. **Comparative analysis focus:** the core of this thesis is solely focused on comparing Firestore and Firebase Realtime database for mobile chat applications.
3. **Key areas scope:** This thesis will only focus on the scalability, real time updates, querying, offline capabilities, and security of both databases.
4. **Geographical scope:** The findings are not limited to any culture, country, or geographical area, the results can be applied across all regions.

The narrow scope allows more detailed and accurate results and is more feasible due to the time and resource constraints.

### 1.5 Constraints

The following highlights what will not be part of the thesis.

1. **Limited to 2 databases:** This thesis only involves Firestore and Firebase Realtime database and doesn't include an analysis of other database technologies.
2. **Exclusion of other mobile app types:** While this thesis has the potential of benefitting a broad range of mobile applications, the research questions were designed to specifically address the most important requirements for mobile chat applications and the thesis will focus only on mobile chat applications. Which means that some findings might need to adapt to different types of applications with different technical requirements.

3. **Security and offline features evaluation:** Due to legal issues, limited knowledge, and time constraints, the security features and offline features will not be tested in the methodology; instead, the evaluation will be based on existing literature.

## 1.6 Structure of the Document

The following presents the sequence and a general explanation of the upcoming chapters.

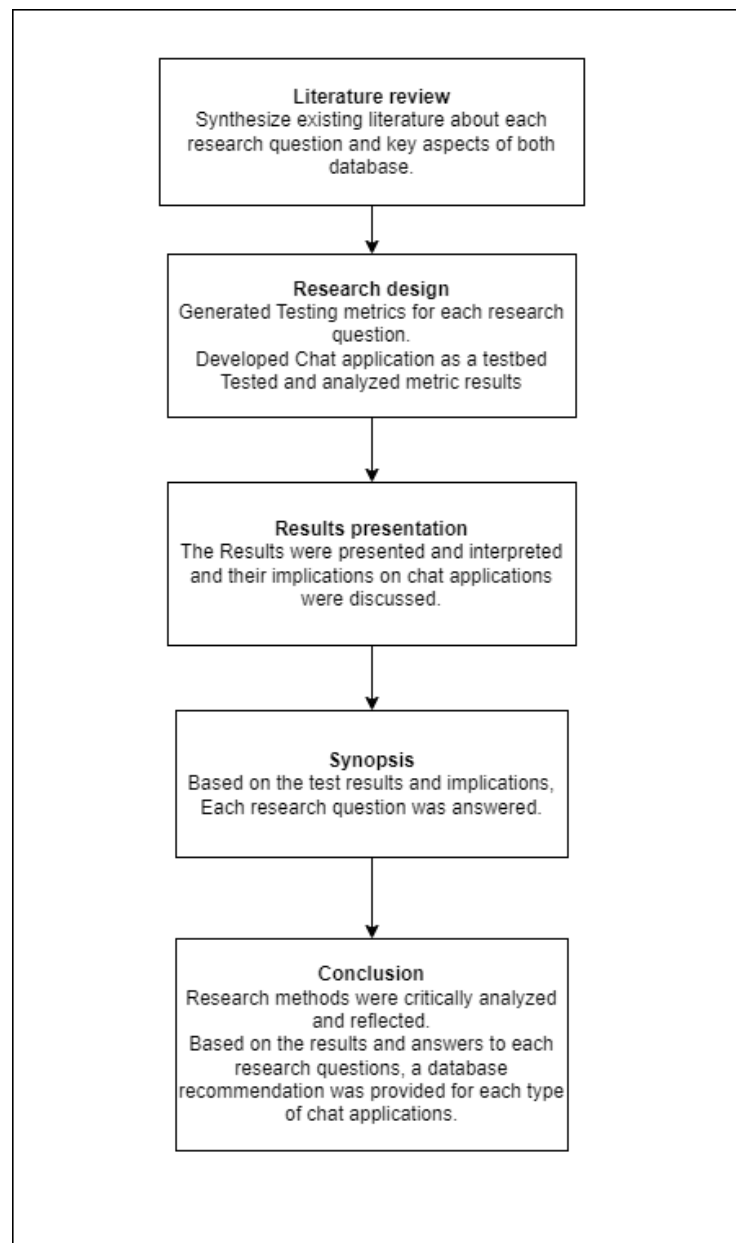


Figure 1. Logical sequence of chapters

## 2 Literature Review

The following presents a synthesis of existing literature on Cloud Firestore and Firebase Realtime database as well as the scalability, real time updates, querying, and security of databases and distributed systems, this synthesis draws from a variety of sources, including peer-reviewed journal articles, case studies, blogs, technical documentation, and existing models published between 2000-2024. The literature was chosen based on its relevance to the core themes and research questions and the credibility of the sources, priority was given to literature that provided a comprehensive analysis or significant empirical findings related to the key concepts.

### 2.1 Theoretical Background

#### 2.1.1 Firebase

Firebase's wide range of services simplify the development and deployment process of mobile chat applications. As per Firebase's official documentation, the relevant services include:

- **Authentication:** Firebase abstracts most of the complex authentication code to provide pre-built authentication methods that simplify the development process. Developers can use these methods to authenticate users via e-mails, phone numbers, or social media accounts.
- **Cloud functions:** Cloud functions allow developers to run backend code when triggered by events generated by http requests or firebase services.<sup>4</sup> Cloud functions can be used to develop custom features for chat applications such as automated responses, chatbots, and message moderating.
- **Cloud messaging:** Cloud messaging allows developers to send messages and notifications from the server to client devices, which makes it particularly useful for chat applications for notifying the users of new messages and updates.

#### 2.1.2 Cloud Firestore and Firebase Realtime Database.

In the upcoming section, we will analyze the underlying concepts, architecture, features, and mechanisms of Firestore and Firebase Realtime database that contribute to their scalability, real-time updates, querying, and security, with the aim of providing a comprehensive theoretical comparison of both databases, which will lay the groundwork for the practical comparison in the research design.

---

<sup>4</sup> (Pandey, 2023, p. 2)

## Cloud Firestore Data Model

Cloud Firestore utilizes a Document Store data model, where data is organized in collections and documents. Related documents are grouped into a collection, and each document in Firestore is identified by a string and contains a set of key value pairs totaling a maximum of 1Mib.<sup>5</sup> Documents can also contain subcollections, forming a nested hierarchy that enables versatile data organization. Figure 1 illustrates how chat application data can be stored in the document store model.

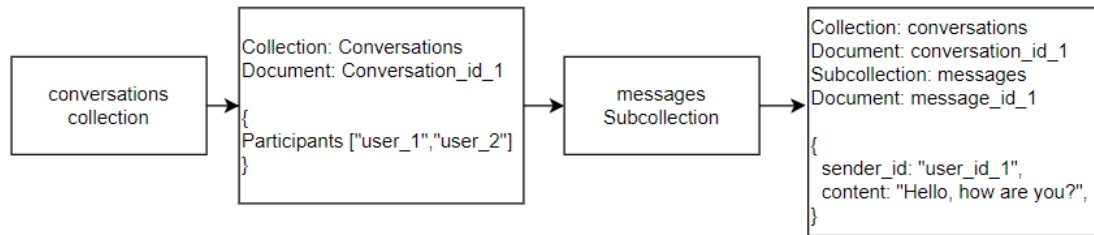


Figure 2. Firestore document store model for chat applications.

## Firebase Realtime Database Data Model

Firebase Realtime database stores data in a JSON document format, JSON documents are essentially collections of key-value pairs, where the value can also be another JSON document.<sup>6</sup> This nesting capability allows storing data hierarchically while keeping the data structure relatively simple.

Firebase Realtime's JSON data structure allows storing many data types including Strings, numbers, Booleans, Dictionaries, arrays, and null values, which is more than enough to accommodate the needs of a chat application. Despite allowing nesting for up to 32 levels, Firebase recommends keeping the data structure as flat as possible as retrieving data will retrieve all its child nodes and granting someone read or write access to a node will grant them access to all its child nodes.<sup>7</sup>

## Scalability

Scalability refers to a system's ability to handle increased workload by expanding resources without compromising performance, one scalability concept that applies to both Firestore and Firebase Realtime is Horizontal scaling, or scaling out, horizontal scaling involves distributing the workload over multiple independent serv-

---

<sup>5</sup> (Kesavan, Gay, Thevessen, Shah, & Mohan, 2023, p. 2)

<sup>6</sup> (Bourhis, Reutter, Suárez, & Vrgoč, 2017, p. 1)

<sup>7</sup> (Google, Structure your database, 2024)

ers to enhance their processing power, unlike vertical scaling, which focuses on installing additional processors and faster memory and hardware into one server.<sup>8</sup> Both Firestore and Firebase Realtime scale horizontally, horizontal scaling can be achieved through many techniques, such as data sharding and replication.

Data sharding, is a data partitioning strategy employed to achieve horizontal scaling, the database is divided into smaller chunks, or shards, which are then distributed across different nodes.<sup>9</sup> Each shard contains the same database schema and table structure but a different subset of the data, which maintains consistency and simplifies management.<sup>10</sup>

Data sharding can improve availability and fault tolerance by adapting a crash failure model, where a faulty node stops sending and responding to requests.<sup>11</sup> Since data is distributed across multiple shards, sharding also enhances read/write throughput if the operation doesn't require data from multiple shards to complete.<sup>12</sup> Data sharding can also improve response time since shards have fewer rows than the entire database which makes search and retrieval faster.<sup>13</sup>

The sharding mechanisms of Firebase Realtime and Firestore differ, Firestore scales automatically to up to 1 million concurrent connections and 10000 writes per second while Firebase Realtime requires manual sharding at 200 thousand connections and 1000 writes per second.<sup>14</sup> Firestore's auto-scaling which includes automatic sharding significantly affects performance. Auto scaling ensures that during peak usage or any sudden spikes, resources are automatically allocated based on the demand.<sup>15</sup> The Firebase Realtime database requires more effort and time from developers due to its manual load balancing and resource allocation process, which can result in less effective performance compared to automatic approaches.

Also, in scenarios where the number of connections and writes are less than the maximum limit, many developers might not feel inclined to manually shard the database, which would result in the database remaining unsharded and degrading the performance compared to Firestore, which would've scaled automatically.

---

<sup>8</sup> (Ali, 2019, p. 5)

<sup>9</sup> (Bagui & Nguyen, 2015, p. 1)

<sup>10</sup> (Novotny, 2023)

<sup>11</sup> (Dang, et al., 2019, p. 2)

<sup>12</sup> (MongoDB, 2023)

<sup>13</sup> (Amazon, 2024)

<sup>14</sup> (Google, Choose a Database: Cloud Firestore or Realtime Database, 2024)

<sup>15</sup> (Bell, 2023)

Data replication involves creating replicas of the same data and distributing them into different nodes which ensures high availability via data redundancy. An example of data replication is creating one or more copies of the same document as a failsafe in case the original document encounters a problem.<sup>16</sup> Those copies are then distributed across multiple nodes either locally or geographically. Replication can be done in two ways, master slave, and peer-to-peer, master slave replication involves a single node which processes all updates that are synchronized to all other nodes, in Peer-to peer, there is no master node, all replicas can handle write requests.<sup>17</sup>

Replication can increase data availability by storing copies at different locations, serving as a disaster recovery mechanism, it also reduces read latency by distributing replicas to multiple locations, enabling users to retrieve data from the nearest site.<sup>18</sup> Replication does little to improve write performance though, in master slave replication, the master node's capacity caps write throughput, while peer-to-peer replication faces challenges with data consistency.<sup>16</sup>

Firestore offers automatic multi-region and regional replication, thanks to its auto scaling, Firebase Realtime on the other hand allows setting only 1 location for each database instance.<sup>19</sup> Firebase Realtime's single region constraints may negatively impact read latency for users far from the location and also contribute to lower availability compared to Firestore, with Realtime database exhibiting a 99.95% uptime compared to Firestore's 99.999% uptime.<sup>14</sup> But while Firestore's Replication improves availability, for a multi-region deployment, it comes with a latency cost as the communication between replicas take longer round trips.<sup>20</sup>

## Querying

Querying in databases can be simple queries, such as reading, inserting, modifying, and deleting entries, or they can be compound queries, which involve retrieving data based on multiple conditions or performing complex aggregations. Simple queries form the backbone of chat applications, as countless messages need to be constantly written and retrieved from the database, compound queries are essential for providing advanced functionalities such as message filtering.

Firestore and Firebase Realtime have different querying capabilities, Firestore's complex data model allows for shallow querying, which means that a document

---

<sup>16</sup> (Mansouri, Javidi, & Zade, 2021, p. 2)

<sup>17</sup> (Costa, Maia, & Carlos, 2015, p. 3)

<sup>18</sup> (Naeem, 2023)

<sup>19</sup> (Google, Select locations for your project, 2024)

<sup>20</sup> (Google, Understand reads and writes at scale, 2024)



can be fetched without having to fetch all of the data in its linked subcollections, which mitigates any unnecessary data retrieval.<sup>21</sup> This makes it very suitable for retrieving data from complex databases. On the other hand, queries in Firebase Realtime tend to retrieve the entire subtree, which leads to retrieving more data than necessary.<sup>22</sup>

For data retrieval, Firestore automatically creates indexes for every field for simple queries.<sup>23</sup> Due to Firestore's automatic indexing for every field, any write operations require making necessary updates to the indexes table, fields being added need corresponding inserts in the index table, fields being deleted need to have their indexes deleted, and fields being modified need both deletion (old values), and insertion (new values) in the indexes table.<sup>20</sup> Firebase Realtime does not require indexes unless the REST API is used, but as the data being queried grows, the performance degrades, so it is important to add indexes if the queried dataset is large, indexes can be created by defining the `indexOn` rule in the security rules and then specifying the path of the data to be indexed<sup>24</sup>

Firestore has much more powerful queries than the Firebase Realtime database for Firebase Realtime, Compound queries can only be done via denormalization, which involves creating another field that combines both queries. For Firestore denormalization isn't needed and, indexes are defined for compound queries instead.<sup>21</sup> Data denormalization involves redundancy, and while denormalization does speed up reads, it slows down writes.<sup>25</sup>, but Firebase Realtime offered a solution for denormalization, called multi-path updates, it allows updating data at multiple locations simultaneously.<sup>26</sup>

For writing data, Firestore offers batch writes, which allow executing multiple write operations that involve any combination inserting, updating, and deleting as a single batch.<sup>27</sup> Batch writes can be useful for chat applications where multiple messages need to be deleted or modified at once without having to execute each operation individually.

---

<sup>21</sup> (Kerpelman, 2019)

<sup>22</sup> (Bigelow, 2019)

<sup>23</sup> (Google, Manage indexes in Cloud Firestore, 2024)

<sup>24</sup> (Google, Index Your Data , 2023)

<sup>25</sup> (Hollingsworth, 2017, p. 19)

<sup>26</sup> (Google, Saving Data, 2024)

<sup>27</sup> (Google, Transactions and batched writes, 2024)

A study by Kot & Smółka<sup>28</sup> measured the CRUD operations execution time for Firebase Realtime, Firestore, and SQLite under simple key-value data schemas, complex data schemas, and large data schemas, they observed that Firebase Realtime Database performed better than Firestore in every aspect except for reading data from a complex data model.

## **Offline Capabilities**

Both Firebase Realtime database and Firestore offer Offline capabilities. When manual disk persistence is enabled in Firebase Realtime, it locally caches a copy of the data that is being actively listened to, queries can be performed on the cached data and the results are reflected post reconnection. Disk persistence has a default threshold of 10MB, after that, it will start to delete older cached data. Firebase Realtime can also detect when clients connect or disconnect.<sup>29</sup> Those offline capabilities apply to both Android and IOS.

Firestore's offline persistence mechanism is different, offline persistence is enabled by default and locally caches every document received by the backend, queries can be performed on the cached data and Firestore also offers an option to automatically add local indexes, when automatic indexing is turned on, it evaluates which collections have the largest number of cached documents and optimizes the local query performance. Firestore also allows manually configuring the cache threshold with a default threshold size of 40MB on android and 100MB on IOS.<sup>30</sup>

Based on the theoretical insights, Firestore clearly offers better offline capabilities for chat applications than Firebase Realtime database. Firestore's automatic local caching of every document as well as the high threshold allows more messages and message history to be available for the user compared to Firebase Realtime's 10MB threshold, automatic local indexing allows for faster retrieval of cached data. Additionally, the manual configuration of the cache threshold allows for control over data storage and management.

Firebase Realtime's offline capabilities, while limited can still provide a solution for chat applications with lower data usage, the ability to detect when a user is offline or online allows the application to act upon those states, such as notifying users when they are offline, and show other users that the user is online.

## **Real-Time Updates and Consistency**

---

<sup>28</sup> (Kot & Smółka, 2023)

<sup>29</sup> (Google, Enabling Offline Capabilities on Android, 2024)

<sup>30</sup> (Google, Access data offline , 2024)

In 2000, Brewer <sup>31</sup> proposed the CAP theorem, which states that a distributed system can only offer two of the following three guarantees:

- Consistency: Every read operation retrieves the most recent write, which means that all nodes must see the same state of the data at the same time.
- Availability: Every request must receive a response, without a guarantee that it contains the most recent write.
- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped by the network between nodes. A partition tolerant system can sustain any amount of network failure that doesn't result in the failure of the entire network.

Since they all cannot be achieved and partition tolerance is the necessary condition expected from the user, there will always be a trade-off between consistency and availability/latency.<sup>32</sup> Eventual consistency prioritizes availability/latency over consistency, and how strong consistency prioritizes consistency over availability/latency.

Firestore offers Strong consistency, which means that all reads return the latest version of the data that reflects all writes that were been committed up until the start of the read.<sup>20</sup> Strong consistency ensures that any changes made must be synchronized across all nodes before being presented by the server, this ensures that all nodes have a consistent view of the data. Firestore achieves strong consistency through many means, it provides Atomicity, Consistency, Isolation, Durability (ACID) properties for all kinds of write operations and also offers transaction serialization which allows transactions to execute in a serializable order. Additionally, Firestore offers synchronous replication, it uses the Paxos algorithm to ensure consistent replication to the different copies, which allows the user to always read latest version of data.<sup>20</sup> While a strongly consistent database can be beneficial for a chat application to ensure that all messages are delivered consistently without data loss and in the correct order without duplication, it comes with its trade-offs, synchronizing updates across multiple servers can lead to increased latency.<sup>33</sup> A strongly consistent database also results in worse availability/latency based on a study conducted by Diogo et al.<sup>34</sup> who studied the consistency models of five different NoSQL databases and found that availability and latency was always compromised when configuring their consistency model to favor strong consistency.

---

<sup>31</sup> (Brewer E. A., 2000)

<sup>32</sup> (Pankowski, 2015, p. 1)

<sup>33</sup> (Ahmad, 2023)

<sup>34</sup> (Diogo, Cabral, & Bernardino, 2019)

In Firebase Realtime, events will always be triggered when local state is changed but those events would reflect the state of the data eventually even when local operations or timing cause temporary differences<sup>35</sup>. This indicates that Firebase Realtime operates on an eventual consistency model. Eventual consistency emphasizes low latency over the risk of displaying stale or outdated data, the data will eventually show once the replicated nodes are synchronized.<sup>36</sup> Lower latency and high availability is guaranteed since the system does not wait for all nodes to be updated before moving forward with operations. A study by Bailis & Ghodsi<sup>37</sup> showed that eventually consistent systems improve availability, latency, and performance at the cost of semantic guarantees. The main downside are temporary inconsistencies, even though Firebase Realtime ensures that writes are written and broadcasted to clients in order, there will be moments where users might see different states of the data, for example in group chats, a message might appear to users immediately after being sent, while for others, it might be delayed, which could disrupt the flow of conversation.

### **Adherence to Standards**

According to the Firebase documentation, Firestore completed the evaluation process for the following International Organization for Standardization (ISO), and System and Organization Controls (SOC) standards: ISO 27001, ISO 27017, ISO 27018, SOC 1, SOC2, and SOC3. Firebase Realtime on the other hand completed the evaluation for ISO 27001, SOC 1, SOC2, and SOC3 standards but not ISO 27017, and ISO 27018.<sup>38</sup> ISO 27017 is derived from ISO 27002 and suggests additional cloud security controls that weren't specified in ISO 27002.<sup>39</sup> ISO 27018 provides guidelines and additional practices for cloud providers to protect Personally Identifiable Information (PII), enhancing ISO 27001's controls and offering additional PII protection advice not found in ISO 27002.<sup>40</sup>

Firebase Realtime's lack of compliance with ISO 27017 and ISO 27018 indicates a narrower scope of cloud-specific security and privacy practices while Firestore's compliance with the additional standards suggest a broader and more detailed commitment to security and privacy.

### **Security Rules**

---

<sup>35</sup> (Google, Retrieving Data, 2024)

<sup>36</sup> (Petrov, 2021)

<sup>37</sup> (Bailis & Ghodsi, 2013)

<sup>38</sup> (Google, Privacy and Security in Firebase , 2024)

<sup>39</sup> (ISMS, ISO/IEC 27017 Cloud Security Controls, 2015)

<sup>40</sup> (ISMS, Understanding ISO 27018:2020, 2020)

Security Rules are fundamental for controlling access to user data, both Firebase Realtime and Firestore allow developers to define rules that determine who can read from and write to the database, the security rules of both databases share some similarities in purpose but are implemented differently, the main difference is that Firestore's rules are non-cascading and combine authorization and validation,<sup>41</sup> Firebase Realtime's cascading security rules stem from its JSON tree data structure. The rules are made up of JavaScript like expressions and stored in JSON format and the structure of the rules must match the database's data structure which means that if a rule grants a read or write access at a specific path, then all child nodes are granted access, validate rules do not cascade and must be satisfied at all levels for a write to be allowed.<sup>42</sup>

Firestore's non cascading rules stem from its document data model and how rules are structured. Firestore's read/write rules consist of a match statement which defines the path of the document, and an allow expression where the condition for reading/writing data is defined, the defined security rule only applies for the matched path and does not apply to any subcollections or documents that stem from the path<sup>43</sup>. Firestore's non cascading security rules give the developers more precise control over access to each document and ensures that no unintended permissions are given to subcollections or documents, which enhances the security, compared to Firebase Realtime which doesn't provide as much control.

Cloud Firestore theoretically provides better security mechanisms than Firebase Realtime database and is preferable choice for chat applications that require a higher level of security, this conclusion stems from Firestore's adherence to a broader range of security standards compared to Firebase Realtime, its more complex and non-cascading security rules which provide more control over data access compared to Firebase Realtime's cascading rules, its multi-region replication option which enhances data availability and disaster recovery which Firebase Realtime does not have, and its overall more complex and organized data structure. Together those aspects contribute to Firestore's superior security mechanisms and measures.

---

<sup>41</sup> (Rai, 2023)

<sup>42</sup> (Google, Learn the core syntax of the Realtime Database Security Rules language , 2024)

<sup>43</sup> (Google, Structuring Cloud Firestore Security Rules, 2024)

### 3 Research Design

The research design was structured to provide a practical evaluation of Firestore and Firebase Realtime database, the approach involved the development of two identical mobile chat applications, one application utilized Firestore as a database and the other utilized Firebase Realtime database. Both applications were then tested across a set of metrics derived from each research question to ensure a comprehensive practical comparison. The research design involved a series of phases that contributed to the overall outcome.

1. **Testing metrics identification:** We began with identifying a one or more testing metrics for each research question, each metric chosen to test a distinct aspect of the question, which ensured that all critical aspects related to each research question were tested.
2. **Requirements gathering:** The requirements of the chat application and databases were then gathered; this phase guaranteed that the applications were developed with identical functionalities with each database configured to utilize all its necessary features, this ensured a full and fair comparison of both databases.
3. **Requirements specification:** The requirements were then listed in a table and categorized into functional, non-functional, and technical requirements with explanations, this documentation acted as a blueprint when developing the applications.
4. **Technology identification:** The relevant frameworks and programming languages for the applications were defined and justified based on their suitability.
5. **Development:** This is when both applications were developed based on the specified requirements.
6. **Testing preparation:** For each metric, the entire testing process was specified and documented, which involves testing tools and technologies, procedures, environments, conditions, variables, repeatability, and any other necessary aspects.
7. **Analysis Techniques and tools:** For each metric, the specific analysis techniques and tools were specified to ensure an accurate and precise evaluation of the collected data.
8. **Testing and analysis:** The applications were tested for each metric and the results were documented and then analyzed.

This approach allowed a comparison that directly reflects each database's performance in real world scenarios, linking theoretical insights that were outlined in the literature review to the practical outcomes by providing direct evidence of the strengths and weaknesses of both databases. The empirical data is critical for developers, as they don't have to rely solely on theoretical insights or comparisons, instead, they're provided with clear, practical insights on the performance of each database under scenarios that resemble their own use cases.

### 3.1 Testing metrics

The metrics for testing each database were chosen, each research question was thoroughly analyzed except for the security question which won't be tested, metrics critical to answering each question accurately and fully were chosen.

#### 3.1.1 Scalability and Real-time updates testing metrics.

The Scalability metric(s) were derived from insights on existing literature that mentions the benefits of horizontal scaling on performance, as well as previous literature cited in the literature review, those insights were then translated into testing metrics specific for chat application features. The metric(s) for real time updates did not require extensive data collection but were derived based on a synthesis of specific sections of the literature review mentioned previously as well as general knowledge of chat applications and both databases. The metrics defined are:

1. **Read response time under different loads:** Measures the response time of messages read from the database under different concurrent user loads.
2. **Write response time under different loads:** Measures the response time of messages written to the database under different concurrent user loads.

#### 3.1.2 Querying metrics

The testing metrics for querying were derived from general knowledge about query operations and previous literature cited in the review such as the study by Kot & Smółka<sup>28</sup> provided some insights on how querying aspects can be measured. The metrics defined are:

1. **Execution time for simple CRUD queries:** Measures the Create, Read, Update, Delete (CRUD) query execution time under different conditions.
2. **Execution time for compound queries:** Measures the execution time of queries that require filtering multiple fields at once, the query will be measured by filtering all messages in a chat room from a specific user in a specific timeframe. This metric is helpful to determine if denormalization in Firebase Realtime database is worth the added effort and complexity.

### 3.2 Requirements Engineering

The General requirements for the study were divided into three types, functional, non-functional, and technical requirements. Functional requirements describe the specific features, behaviors, and actions that a software system should provide, they're directly related to the functionalities that the end users can perform. To ensure valid testing, it was crucial that our application mirrored real world scenarios and offered the same critical functionalities that all chat applications offer. The Functional requirements were gathered by observing and documenting the features of multiple popular chat applications, those applications include WhatsApp, Messenger, Viber, and Slack. The critical features that were prominent in all applications were considered critical and necessary for our testing application.

Non-functional Requirements describe the quality attributes of software systems, for this study, the non-functional requirements were derived from general knowledge of software testing best practices and both databases.

Technical Requirements are the technical specifications and capabilities that a system must possess to function as intended. For this study, the technical requirements were generated to ensure that the manual design of one database doesn't give it an unfair performance advantage over the other when testing. The technical requirements were mainly derived based on knowledge from prior testing and information from the literature review conducted in the previous section, which highlights all necessary aspects of both databases.

### 3.3 Requirements Specification

The following requirements were deemed necessary for the study and include front-end, database, and testing requirements.

Table 1. Requirements for Chat Application and testing

Requirement type	Requirement	Reason	Scope
Non-Functional	Both Chat applications must have fully identical Front ends.	To ensure that any differences in performance is attributed solely to the databases, not the UI design.	Front-end
Non-Functional	The Schema of both databases must be structured as efficiently as possible, following the Firebase documentation's best practices.	To ensure that the performance differences between the databases aren't attributed to poor design or implementation.	Database



Non-Functional	For each metric, both databases must be tested using the exact same testing tools. Environments, and conditions	To ensure that the comparison is fair, accurate and consistent, and performance differences are not attributed to the use of different testing tools, environments, and conditions.	Database/ Testing
Functional	The applications shall incorporate mechanisms for user authentication.	To allow multiple users to be simulated during the testing.	Back-end
Functional	The applications shall provide functionality for users to send and receive text messages.	For testing the scalability and real time updates of both databases.	Front-end/Cloud Storage
Functional	The applications shall allow users to initiate and manage multiple chat sessions simultaneously.	To ensure a realistic testing environment	Front-end/Database
Technical	Each message sent shall be timestamped to record the exact time of sending and receive.	For measuring the latency time for real time message updates.	Database
Technical	Both databases shall implement proper indexing strategies to ensure efficient data retrieval. For Firestore, this includes the creation of composite indexes for compound queries, while Firebase Realtime Database will be optimized within its indexing capabilities.	To ensure fair and accurate testing of each database's querying capabilities for chat applications by taking full advantage of each database's querying features.	Database
Technical	Firestore shall be configured for multi-regional replication.	This is done to leverage Firestore's full capabilities ensuring that the testing encompasses Firestore's advanced replication features.	Database (Firestore)
Technical	The physical locations of both Firestore and Firebase Realtime Database instances shall be set in Europe.	Since latency is impacted by distance, this ensures that one database doesn't have an unfair latency advantage over the other as the location is manually set by the developer.	Database
Technical	The application shall utilize the most recent versions of both Firestore and Firebase Realtime Database.	To make sure that one database doesn't have an advantage over the other simply because it is a newer/different version.	Database

### 3.4 Technology

The following section outlines the programming languages, frameworks, technologies, and tools used for the project. For front-end development, the Flutter framework was used which utilizes the Dart programming language, Flutter was chosen due to its exceptional support for building cross platform mobile applications, and since both Flutter and Firebase are Google products, they integrate very well with each other with the help of the FlutterFire plugin which provides the Flutter application access to many Firebase services such as Firebase Authentication, Cloud storage, Cloud functions, and Cloud messaging. The firebase services used were Firebase Authentication to authenticate users, and Cloud messaging for message notifications.

For Testing, Firebase Cloud functions was used, which allowed writing NodeJS scripts that automated the processes of reading and writing database entries, and allowed monitoring query execution times, which was necessary for the query performance testing.

To simulate user load, JMeter was chosen to simulate requests to Firestore and Firebase Realtime's REST API endpoints, JMeter was chosen due to its low learning curve and extensive documentation and community support available due to its popularity.

### **3.5 Development and testing**

The testing was divided into two parts, the first part involved testing the metrics for the scalability and real-time updates, which required the application to be load tested, and the second part involved testing the querying metrics which did not require load testing the application and focused on server-side querying.

#### **3.5.1 Database models and configuration**

Firestore's database instance location was set to eur3 multi-region, which replicated the instances across Netherlands and Belgium for reads/writes, and Finland as a witness region. Firebase Realtime's instance was set to Europe-west1 in Belgium which ensured that both databases were in the same continent. The front-end for the chat application was then created for Firestore first, and the application was copied and configured to connect to Firebase Realtime database, to ensure that the front ends were completely identical.

Due to time constraints, the structure of both databases was simple and only focused on essential chat application components. Firestore's model consisted of a Users collection for user documents, and a Chat rooms collection for chat room documents, chat room documents identified each chat room, and each individual chat room document contained a messages subcollection which stored documents for

the messages in the room. Firebase Realtime was structured in a similar way with a few differences. It consisted of 2 root nodes, the user node which stored user information, and the chat room node, which contained details about the chat room, each chat room contained a messages node, which contained details for each message.

During the query testing, a messagesBySender node, which contained all messages sent by each sender was added as a child for each chat room. The messagesBySender node was needed to test the execution time of the compound query, as Firebase Realtime doesn't directly support compound queries and needed denormalization.

### 3.5.2 Query Testing Process

For Testing the simple query and compound query execution time we had to ensure fairness, consistency, and accuracy, The execution time for simple queries involved testing the CRUD execution time for messages, and for the compound query test, it involved retrieving messages in different volumes from a chat room after filtering by sender and timestamp.

Cloud functions were utilized to execute server-side queries from our client device. This became necessary due to our distant physical location, which introduced added latency for queries and rendered any server-side queries directly from our local device useless. By deploying cloud functions in the same geographical region of the databases, we were allowed to populate the databases, execute all necessary queries, and retrieve the execution time directly from Google's consistent network without the added latency and bandwidth inconsistencies of a distant physical location. Each operation aimed for a realistic interpretation of a chat application operation.

- **Create:** As messages are regularly written in chat applications, The execution time for inserting 1, 100, and 1000 message records was measured for both databases,
- **Read:** As messages are frequently read in chat applications when entering a chat room or when receiving new messages, The execution time for reading the most recent 10, 100, and 100 messages from a chat room was measured for both databases.
- **Update:** As updates to messages such as editing message content and marking the message as unread is common in chat applications, the execution time for updating the read statuses for 1, 100, and 1000 messages was measured for both databases.
- **Delete:** As messages are regularly deleted individually or in bulk in chat applications, 1, 100, and 1000 messages were deleted from the database and the execution times were recorded.

- **Compound query:** As chat applications often offer message filtering functionalities, the execution time for filtering and retrieving messages by sender and timestamp was measured for the retrieval of 10, 100, and 1000 message documents. In order to execute the query in Firestore, a composite index was created on the sender and timestamp field, for Firebase Realtime, denormalization converted the compound query to a simple filter query, the messages were retrieved from the messagesBySender node by filtering the timestamp field.

The results of each query were then placed in a separate table, highlighting the average execution time in milliseconds for every volume of records. For analysis, apart from which database is faster, the execution time increase in percentage for each query executed from the lowest record count to the highest will be measured to determine which database is more consistent as the volume increases.

### **Error assessment**

To ensure that the results were precise, each query execution was repeated three times initially, to mitigate any cold start effects or overhead from initial setup, following this, they were repeated five more times, and the average of the five executions was calculated, this allowed us to minimize variability and inconsistencies in execution times. Since both databases cache frequently retrieved data for faster retrieval, and reading the same data five times will surely cache it, operations that required data retrieval were executed on different data every time, this ensured that operations were not influenced by caching effects and provided a more accurate representation of query performance.

To further ensure that the results were accurate, all operations were conducted on different message volumes, representing the different number of records operated on. At each volume level, the test was conducted on small (100bytes), medium, (500bytes), and large(1KB) messages content sizes, with the average of all sizes calculated to derive a final result for each message volume level. Additionally, to ensure that both databases utilized their full capabilities and make the comparison as accurate as possible, Firestore utilized its batch writing capabilities for Create, Update, and Delete operations on 100 and 1000 records, and Firebase Realtime database utilized multi path updates to ensure write efficiency in its denormalized form.

Create, Update, Delete, and the compound query were conducted on Firestore, Firebase Realtime with denormalization, and Firebase Realtime with client-side filtering instead of denormalization, this allowed us to assess the effect of denormalization on Firebase Realtime and determine if it is worth it or is client-side filtering more efficient. The simple Read operation is not affected by client-side filtering or denormalization so it wasn't included in this comparison.

The entire testing environment and conditions, which include the records operated on and the message content sizes were fully identical for both databases, to ensure that the results are unbiased. The final results were compared with the study by Kot & Smółka<sup>28</sup>, while the nature of the studies may differ in certain aspects and purposes, the study still provided a valuable reference point for validating the accuracy of our findings.

### **3.5.3 Scalability Testing**

Scalability testing was conducted to evaluate the performance of Firestore and Firebase Realtime Database under increasing loads and demands, the testing scenario involved simulating 400, 800, 2000, and 5000 users using JMeter with half of the users performing one write operation per second and the other half performing one read operation every second. Each test was five minutes long due to resource constraints as Firestore charges for writes and reads and firebase Realtime charges for data uploaded and downloaded. After the tests for each load, all messages from all rooms were deleted to prepare for the next load. Each user logged in, entered the room, and wrote or read 1 message per second, until the test was done.

#### **What was needed for the testing.**

1. Database population: Firebase AdminSDK was used to populate both databases with the necessary number of users and chat rooms.
2. Comma Separated Value (CSV) files: one CSV file was needed for user authentication data for every user, one CSV file was needed for writing messages, it contained a list of chat room ID's that were populated, and all messages to be written in each chat room, and another CSV file was needed for reading data which only contained a list of all chat room ID's. This was needed so the simulated users would know which chat rooms to read and write from.
3. Access tokens: For the users to be able to log in, access tokens were generated by firebase auth AdminSDK
4. Ethernet cable: To ensure network consistency and that any differences in response time are fully attributed to the databases and not to temporary network inconsistencies.

After each test, the average response time was generated, each test load was repeated three times with the average of the three being the final result, this helped mitigate any anomalies.

#### **Constraints**

There were many aspects that prevented the test from being utilized to the full.

1. Resource allocation: A maximum of 5000 concurrent users were allowed within the testing environment due to hardware constraints which included CPU and Memory, any users above 5000 caused JMeter to crash.
2. Cost considerations: More Tests would have conducted with longer durations but Both databases charge on a Pay-as-you-go model and simulating more users would have been costly.
3. Time constraints: due to limited time, more complex requests could not have been done.
4. Reporting limitations of testing tools: While JMeter offers reporting for throughput, response time, and error rates, monitoring other metrics was only possible for each individual request, not the test average.



## 4 Results & Discussion

Since the testing was divided into Query testing and scalability testing, this chapter will be divided into two subchapters, and each subchapter will be for presenting, interpreting, and discussing the results of one of the aspects. While Scalability and querying are interconnected, and can be discussed and interpreted in one subchapter, they separated for a focused discussion on each aspect.

### 4.1 Query Testing Primary and Derived Data

Table 2. Execution time for CREATE queries.

Database	Number of records		
	1	100	1000
Firestore	36.6	310.2	2522.2
Denormalized Realtime	8.6	199	1396
Normal Realtime	5	977	6755
Average execution time (milliseconds)			

Table 2 shows that initially, the normal Firebase Realtime exhibited the fastest execution time for Creating 1 record, but as the number of records increased, the execution time increased exponentially with an 135000% increase in execution time relative to the number of records increased and exhibited a much higher execution time than Firestore and its Denormalized form. Firestore presented a more consistent performance than both Firebase Realtime's forms, with an increase of 6790.3%, but was slower for small record counts. The Denormalized Realtime had an increase of 16132.5%, while not as consistent as Firestore, exhibited a much lower execution time than Firestore and its normal form as the records increased.

Table 3. Execution time for READ queries.

Database	Number of records		
	10	100	1000
Firestore	96.8	433.2	2096
Firebase Realtime	8	50	523
Average execution time (milliseconds)			



Table 3 shows that Firebase Realtime is clearly much faster at message retrieval than Firestore at all record numbers, exhibiting almost 4 times less execution time for 10 and 100 records and 4 times less for 1000 records. While Firestore exhibits slower data retrieval, its execution times are more consistent with a 2065% increase compared to Firebase Realtime's 6437.5%.

Table 4. Execution time for UPDATE queries.

Database	Number of records		
	1	100	1000
Firestore	57.4	493	3659.6
Denormalized Realtime	13	82	560.4
Normal Realtime	10.6	241.4	2542.6

**Average execution time (milliseconds)**

Table 4 shows that Firestore is the slowest for all record counts, it also showed an increase of 6275.6%, Firebase Realtime's Denormalized form is the fastest for all record numbers and also the most consistent with an increase of 4210.7%, the Normal form of Realtime database exhibited much faster execution times for updates than for writes but remains slower than its Denormalized form for smaller record counts. Table 4 also showed that UPDATE queries are much faster than CREATE queries for both forms of Firebase Realtime but slower for Firestore.

Table 5. Execution time for DELETE queries.

Database	Number of records		
	1	100	1000
Firestore	45	440.8	2837
Denormalized Realtime	5	126.7	1011
Normal Realtime	8	318	4439

**Average execution time (milliseconds)**

Table 5 shows that the Denormalized form of Firebase Realtime is executes DELETE queries faster than Firestore, but Firestore Is the most consistent with an increase of 6204.4%, compared to Denormalized Realtime's 20120% and Normal Realtime's 55387.5%

Table 6. Execution time for filtering by sender and timestamp.

Database	Number of records		
	10	100	1000
Firestore	80	322	2594
Denormalized Realtime	5	58	520
Normal Realtime	8	105	956

**Average execution time (milliseconds)**

Table 6 Shows that for retrieving records filtered by message and timestamp, Firestore is the slowest but most consistent with a 3142.5% increase, Denormalized Realtime is the fastest but is much less consistent with a 10300% increase, and Normal Realtime database with client-side filtering is the least consistent with a 11850% increase.

In summary, the final results of all operations show that Firebase Realtime's offers faster execution times for all operations in all defined record sizes, with Firebase Realtime's normal form being the fastest for small record counts, while the Denormalized form is the fastest for larger record counts, Firestore is more consistent in execution times as the records increase but seems to have a weakness for update operations.

## 4.2 Query Discussion

Understanding the difference between the theoretical querying capabilities and performance metrics of firebase Realtime and Firestore is important, the statement by Kerpelmann<sup>21</sup> mentioned that Firestore has more powerful querying capabilities than firebase Realtime, due to its built-in support for compound queries while Firebase Realtime requires denormalization.

While this is true to some extent, the superiority of querying capabilities does not translate into overall performance, and do not hold up in the context of Chat applications, the evidence from our test results not only prove that Firestore has worse querying performance for both simple and compound queries, it also highlights how practically efficient Firebase Realtime is when adopting a denormalized form. While Firebase Realtime's Denormalization can introduce added complexity for larger, more demanding applications and Firestore's capabilities might be a solution, this is not the case for chat applications.

Additionally, while Firestore's advanced capabilities can be very beneficial in many scenarios, can introduce challenges in others. A significant aspect of this is Firestore's automatic indexing, while it allows powerful queries, it introduces overhead

for inserts, deletes, and especially updates.<sup>23</sup> The overhead became evident as Firestore's CREATE, UPDATE, and DELETE queries were often slower for all record counts. What was particularly revealing of Firestore's indexing overhead was the execution time for the update queries, despite Firestore utilizing batch writes, the execution time of both forms of Firebase Realtime database for UPDATE queries was lower for all record counts, unlike the CREATE queries where Firestore's batch writes gave an edge over Firebase Realtime's Normal form at higher record counts. This is due to the fact that Update queries require two operations, the old index must be deleted, and the new index must be added, which doubles the overhead, unlike the CREATE and DELETE queries which only require one operation.<sup>24</sup>

This overhead significantly impacts Firestore when used for chat applications as Firestore is slower for smaller record counts and chat applications usually have a continuous stream of independent messages being written, updated, or deleted with each message requiring a separate query. Even for instances where a large number of messages need to be queried at once, for example: mass deleting or updating a large number of messages, Firestore is still slower when executing larger record counts than Firebase Realtime's Denormalized form for all queries and is also slower than Firebase Realtime's Normal form for UPDATE queries and compound queries even though Firebase Realtime's Normal form utilized client-side filtering.

Firebase Realtime's multi path updates massively boosted the write query execution time for Firebase Realtime's denormalized form, which means that Firebase Realtime, when denormalized, allows performing both simple and compound queries at a much faster rate than Firestore with the only disadvantage being the added complexity of manually denormalizing the database.

Despite Firebase Realtime having lower execution time for all operations, it did not exhibit Firestore's levels of consistency in execution times, for Firebase Realtime, the performance degrades as the data being retrieved grows which makes defining indexes necessary for large datasets.<sup>24</sup>, but even with the necessary indexes defined, both Forms of Firebase Realtime encountered challenges in maintaining execution time consistency for read operations, exhibiting a more exponential growth.

While Firestore offers better consistency as the number of queried records grows, the trade-off in execution time makes it less suitable for applications like chat applications. In these cases, where traffic consists of a large number of independent writes and reads, the speed of queries is prioritized over consistency, making Firestore's advantages less impactful.

### 4.3 Scalability Primary and Derived Data

The following presents the test results from the response time testing, it contains the average read and write response times for 400 users, 1000 users, 2000 users, and 5000 users.

Table 7. Read response time.

Database	Concurrent Users			
	400	1000	2000	5000
Firestore	101	105	129	175
Firebase Realtime Database	91	95	112	371

#### Average read response time (milliseconds)

Firebase Realtime database exhibits overall lower response time for reads for 400,1000,2000, users, but spikes in response time at 5000 users, Firebase Realtime exhibits an average response time increase of 84.51%, while Firestore exhibits an average response time increase of 20.83%.

Table 8. Write response time.

Database	Concurrent Users			
	400	1000	2000	5000
Firestore	105	120	144	204
Firebase Realtime Database	98	105	122	367

#### Average write response time (milliseconds)

Just like the reads, Firebase Realtime exhibits overall lower response time than Firestore, but spikes at 5000 users, Firebase Realtime exhibits an average response time increase of 74.72%, while Firestore exhibits an average response time increase of 25.32%.

### 4.4 Discussion for Scalability

While it's odd that Firebase Realtime Database exhibits such a sudden spike in response times at 5000 concurrent users compared to Firestore which stays relatively stable, it can be attributed to multiple aspects. Data sharding increases throughput<sup>12</sup> and response time<sup>13</sup>, and Firestore automatically scales which includes auto sharding<sup>14</sup>, Meanwhile, Firebase Realtime doesn't shard automatically, so by the time the application reached 5000 concurrent users, Firestore would have scaled automatically but firebase real time would have stayed the same as it was from the start, another reason would be that Firebase Realtime has a maximum write throughput

of 1000/second<sup>14</sup>, and reaching 5000 users would have definitely hit the threshold, which could have performance implications.

While Firebase Realtime experiences a notable increase in response times at 5000 users, for lower user loads it has a slightly lower response time, this is attributed to Firestore's multi region replication, which can introduce additional latency as the number of round trips increases.<sup>20</sup> and also considering that Firestore is Strongly consistent, and strongly consistent databases tend to prioritize consistency over availability and latency.<sup>34</sup>

While it may seem that the combination of strong consistency and multi region replication is a recipe for high write latency due to the increased number of nodes that the data must traverse and synchronize for every write, the test results only show a slight difference in write and read response time, and while response time does not equal latency, latency still makes up a significant part of the response time. So, in comparison, Firestore enjoys all benefits of strong consistency, with only a slight increase in response time compared to Firebase Realtime that enjoys only a slight decrease in response time at the cost of scalability. Even though chat applications are real-time, and require the highest possible response time, the trade-off is worth it for larger applications. While the study Bailis & Ghodsi<sup>37</sup> conducted where they found out that eventually consistent databases improve availability and latency at the cost of semantic guarantees is correct, in the case of Firestore and Firebase Realtime the difference is minimal and not worth compromising scalability.

## 4.5 Synopsis

The study determined which of Cloud Firestore or Firebase Realtime database is more suitable for chat applications that Prioritize either Scalability, Real-time updates, Offline capabilities, Querying, or Security

### 6. Which Database scales more effectively as the number of users and messages increases?

It was found that Cloud Firestore scales more effectively as the number of users and messages increases than Firebase Realtime Database.

### 7. Which database demonstrates more efficient real time capabilities for mobile chat applications?

It was found that Firebase Realtime Database provides more efficient real time capabilities for mobile chat applications than Cloud Firestore.

### 8. Which database offers better offline capabilities for mobile chat applications?

It was found that Cloud Firestore provides better offline capabilities for mobile chat applications than Firebase Realtime Database

**9. Which database offers more efficient querying capabilities for mobile chat applications?**

It was found that Firebase Realtime Database provides more efficient querying capabilities for mobile chat apps than Cloud Firestore

**10. Which database provides more effective security measures and mechanisms to protect user data?**

It was found that Cloud Firestore provides more effective security measures and mechanisms to protect user data than Firebase Realtime Database.

## 5 Conclusion

This study embarked on a comprehensive comparison between Firestore and Firebase Realtime Database to determine their suitability for mobile chat applications, with different requirements, this study's purpose was to understand the suitability of both databases focusing on their scalability, real-time updates, offline capabilities, querying, and security measures.

Firestore is better suited for complex mobile chat applications with larger user bases due to its superior scalability, robust offline capabilities, and advanced security mechanisms. These features make Firestore a more reliable choice for larger user bases, which are more likely to encounter security threats. On the other hand, Firebase Realtime's superior real time capabilities and querying make it the optimal choice for chat applications with smaller user bases that usually prioritize instantaneous message delivery over scalability and security.

### 5.1 Critical Reflection

The study acknowledges limitations, the most notable one was the focus on very broad and simple chat application scenarios when developing the database and testing. Which did not encompass complex or very realistic scenarios. This made them difficult to interpret and discuss in the context of real-world chat applications.

### 5.2 Recommendations for Future Research

For methodologies involving software testing, it's important to structure the tests to mirror real-life scenarios. While creating realistic scenarios can be challenging, it is essential, as tests that don't mirror real-world conditions may compromise the validity of the results

## 6 References

- Ahmad, A. (2023, June 9). *Consistency Patterns in Distributed Systems: A Complete Guide*. Retrieved from designgurus: <https://www.designgurus.io/blog/consistency-patterns-distributed-systems>
- Ali, A. H. (2019). A survey on vertical and horizontal scaling platforms for big data analytics. *International Journal of Integrated Engineering*, 11(6), 138-150.
- Amazon. (2024). *What is Database Sharding?* Retrieved from Amazon AWS Documentation: <https://aws.amazon.com/what-is/database-sharding/>
- Bagui, S., & Nguyen, L. T. (2015). Database sharding: to provide fault tolerance and scalability of big data on the cloud. *International Journal of Cloud Applications and Computing (IJCAC)*, 5(2), 36-52.
- Bailis, P., & Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5), 55-63.
- Bell, H. (2023, September 29). *What is Auto-scaling?* Retrieved from Noname Security: <https://nonamesecurity.com/learn/what-is-auto-scaling/>
- Bigelow, S. J. (2019, August 9). *Compare Google Cloud Firestore and Firebase Realtime Database*. Retrieved from Techtarget: <https://www.techtarget.com/searchcloudcomputing/tip/Compare-Google-Cloud-Firestore-and-Firebase-Realtime-Database>
- Bourhis, P., Reutter, J. L., Suárez, F., & Vrgoč, D. (2017, May). JSON: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, 123-135.
- Brewer, E. A. (2000). Towards robust distributed systems. *Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Portland, Oregon.
- Brewer, E. A. (2000, July). Towards robust distributed systems. *PODC*, 7(10.1145), 343477-343502.
- Costa, C. H., Maia, P. H., & Carlos, F. (2015, April). Sharding by hash partitioning. In *Proceedings of the 17th International Conference on Enterprise Information Systems*, 1, 313-320.
- Dang, H., Dinh, T. T., Loghin, D., Chang, E. C., Lin, Q., & Ooi, B. C. (2019, June). Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, 123-140.
- Diogo, M., Cabral, B., & Bernardino, J. (2019). Consistency models of NoSQL databases. *Future Internet*, 11(2), 43.
- Dixon, S. (2023, August 29). *Most popular global mobile messenger apps as of January 2023, based on number of monthly active users*. Retrieved from Statista: <https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/>
- Dudjak, M., & Martinović, G. (2020). An API-first Methodology for Designing a Microservice-based Backend as a Service Platform. *Information Technology and Control*, 49(2), 206-223.
- Google. (2023, February 6). *Index Your Data* . Retrieved from Firebase documentation: <https://firebase.google.com/docs/database/security/indexing-data>



- Google. (2024, February 15). *Access data offline* . Retrieved from Firebase documentation: [https://firebase.google.com/docs/firestore/manage-data/enable-offline#swift\\_1](https://firebase.google.com/docs/firestore/manage-data/enable-offline#swift_1)
- Google. (2024, February 7). *Choose a Database: Cloud Firestore or Realtime Database*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/database/rtdb-vs-firestore>
- Google. (2024, February 15). *Enabling Offline Capabilities on Android*. Retrieved from Firebase documentation: <https://firebase.google.com/docs/database/android/offline-capabilities>
- Google. (2024, March 4). *Learn the core syntax of the Realtime Database Security Rules language* . Retrieved from Firebase Documentation: [https://firebase.google.com/docs/database/security/core-syntax#read\\_and\\_write\\_rules\\_cascade](https://firebase.google.com/docs/database/security/core-syntax#read_and_write_rules_cascade)
- Google. (2024, February 8). *Manage indexes in Cloud Firestore*. Retrieved from Firebase documentation: <https://firebase.google.com/docs/firestore/query-data/indexing>
- Google. (2024, February 8). *Perform simple and compound queries in Cloud Firestore*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/firestore/query-data/queries>
- Google. (2024, March 4). *Privacy and Security in Firebase* . Retrieved from Firebase Documentation: [https://firebase.google.com/support/privacy#firebase\\_data\\_processing\\_and\\_security\\_terms](https://firebase.google.com/support/privacy#firebase_data_processing_and_security_terms)
- Google. (2024, February 21). *Retrieving Data*. Retrieved from Firebase documentation: <https://firebase.google.com/docs/database/admin/retrieve-data#section-event-types>
- Google. (2024, March 4). *Saving Data*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/database/admin/save-data>
- Google. (2024, March 4). *Saving Data*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/database/admin/save-data>
- Google. (2024, February 6). *Select locations for your project*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/projects/locations>
- Google. (2024, January 18). *Structure your database*. Retrieved from Firebase documentation: <https://firebase.google.com/docs/database/web/structure-data#:~:text=All%20Firebase%20Realtime%20Database%20data,structure%20with%20an%20associated%20key.>
- Google. (2024, March 5). *Structuring Cloud Firestore Security Rules*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/firestore/security/rules-structure>
- Google. (2024, March 14). *Transactions and batched writes*. Retrieved from Firebase documentation: <https://firebase.google.com/docs/firestore/manage-data/transactions#batched-writes>
- Google. (2024, February 21). *Understand reads and writes at scale*. Retrieved from Firebase Documentation: <https://firebase.google.com/docs/firestore/understand-reads-writes-scale#:~:text=This%20strong%20consistency%20means%20that,the%20start%20of%20the%20read.>
- Hollingsworth, M. (2017). *Data normalization, denormalization, and the forces of darkness. V.0.4, ed.*
- ISMS. (2015). *ISO/IEC 27017 Cloud Security Controls*. Retrieved from Information Security Management System: <https://www.isms.online/iso-27017/>

- ISMS. (2020). *Understanding ISO 27018:2020*. Retrieved from Information Security Management System: <https://www.isms.online/iso-27018/>
- Kerpelman, T. (2019, January 31). *Cloud Firestore vs the Realtime Database: Which one do I use?* Retrieved from Firebase blog: <https://firebase.blog/posts/2017/10/cloud-firestore-for-rtdb-developers/#better-querying-and-more-structured-data>
- Kesavan, R., Gay, D., Thevessen, D., Shah, J., & Mohan, C. (2023). Firestore: The NoSQL Serverless Database for the Application Developer.
- Kot, S., & Smolka, J. (2023). A performance analysis of a cloud database on mobile devices. *Journal of Computer Sciences Institute*, 29, 360-365.
- Lien, J. (2023, February 17). *What are the disadvantages of database indexes?* Retrieved from Planetscale: <https://planetscale.com/blog/what-are-the-disadvantages-of-database-indexes>
- Mansouri, N., Javidi, M. M., & Zade, B. M. (2021). Hierarchical data replication strategy to improve performance in cloud computing. *Frontiers of Computer Science*, 15, 1-17.
- MongoDB. (2023). *Database Sharding: Concepts and Examples*. Retrieved from Mongo DB documentation: <https://www.mongodb.com/features/database-sharding-explained>
- Naeem, T. (2023, December 1). *What Is Data Replication and How Does It Impact Your Business?* Retrieved from Astera Blog: <https://www.astera.com/type/blog/data-replication/>
- Novotny, J. (2023, October 23). *Database Sharding: Concepts, Examples, and Strategies*. Retrieved from Linode Web Site: <https://www.linode.com/docs/guides/sharded-database/>
- Pandey, R. (2023). Chat Application using React.js and Firebase. *Amity Journal of Computational Sciences*, 7(1).
- Pankowski, T. (2015). Consistency and availability of Data in replicated NoSQL databases. In *2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)* (pp. 102-109). IEEE.
- Petrov, P. (2021, February 18). *An Introduction to Eventual Consistency*. Retrieved from codecoda: <https://codecoda.com/en/blog/entry/an-introduction-to-eventual-consistency>
- Rai, A. (2023, May 15). *Google Cloud Firestore vs Firebase Realtime Database*. Retrieved from Economize blog: <https://blog.economize.cloud/firestore-vs-firebase/>
- Tram, M. (2019). Firebase.





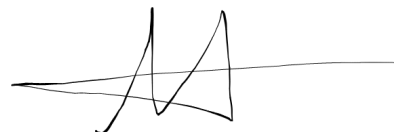
## Declaration of Authenticity

I hereby declare that I have completed this Bachelors/Master's thesis on my own and without any additional external assistance. I have made use of only those sources and aids specified and I have listed all the sources from which I have extracted text and content. This thesis or parts thereof have never been presented to another examination board. I agree to a plagiarism check of my thesis via a plagiarism detection service.

Amman, 20-03-2024

---

Place, Date



---

Signature