ECOLE POLYTECHNIQUE DE BRUXELLES

2024/2025

**INFO-H-415 : Advanced Databases**

# Benchmarking of E-commerce application

Akli-kodjo-Mensah Gloria 000616562
Salma Namouri 000604449
Amélie Liesenborghs 501159

Professor : Esteban Zimányi
Teacher Assistant : Boris Coquelet

2024

# Table des matières

# List of abreviation

**API** :Application Programming Interface

**BSON** :Binary JSON

**CSS** :Cascading Style Sheets

**DB** : Database

**DBMS** : Database Management System

**HTML** :HyperText Markup Language

**JS** :JavaScript

**JSON** :JavaScript Object Notation

**KPI** :Key Performance Indicator

**NoSQL** :Not only Structured Query Language

**RT** :Real Time

**RTDB** :Real Time Database

**SDK** :Software development kit

**UML** :Unified Modeling Language

**UI** :User Interface

**UE** :User Experience

# 1 Introduction

Document stores have emerged as a popular choice for managing semi-structured data due to their flexibility in schema design, support for nested data structures, and ability to scale horizontally (capable of handling large volumes of data by distributing it across multiple servers or nodes). Firebase and Firestore, two cloud-based document databases offered by Google, are widely recognized for their ease of use and robust performance in real-time and scalable environments.

This project assesses the suitability of document stores for our specific application by comparing Firebase and Firestore. The purpose is to see whether the two tools offer significant advantages by providing an analysis, demonstrating their strengths and weaknesses through benchmarking their performance against a traditional relational database (PostgreSQL) on datasets ranging from 1,000 to 128,000 records.

Our study is restricted to Firebase and Firestore only and the benchmarks are specific to a chosen case of ecommerce application, which may limit the generalization of our findings to other scenarios. Despite these limitations, the project aims to deliver valuable insights into the comparative strengths and weaknesses of document stores.

# 2 Document Stores

Document-oriented databases, often referred to as document stores, are a type of NoSQL database that store, manage, and retrieve data as documents. Unlike traditional key-value databases, document stores structure data within each document in a specific way, allowing for more complex and hierarchical representations of data. Each document is identified by a unique key and contains its associated data, which can be structured or semi-structured.

## 2.1 Definition and Characteristics

Document stores organize data in a tree-like hierarchy where each document represents a record. These documents are flexible in format and may include complex data structures such as arrays and nested objects.

Key characteristics of document-oriented databases include :

— **Data Representation** : Data in document stores is typically encoded in formats like JSON (JavaScript Object Notation), BSON (Binary JSON), or XML. These formats allow for easy data interchange and hierarchical representation.

— **Scalability** : Document databases are designed for horizontal scalability, distributing data across multiple servers or nodes to handle large-scale workloads efficiently.

— **Flexibility** : The schema-less characteristic and the hierarchically stored data allow developers to model complex real-world relationships without requiring multiple tables or joins, as in relational databases.

— **Querying** : Document stores often include powerful query capabilities, enabling operations like filtering, aggregation, and full-text search. Query languages such as JavaScript, Python, or database-specific APIs are commonly used.

## 2.2 Advantages

— **Flexibility** : The schema-less structure allows developers to adapt quickly to changes in data requirements, adding new fields without modifying existing documents.

— **Performance for Certain Queries** : Since data is stored in a denormalized format, where data is intentionally duplicated or combined rather than being split across multiple related tables in a normalized form, document stores excel in read-heavy operations and applications where queries retrieve complete documents.

— **Scalability** : Horizontal scaling enables the storage and management of massive datasets, making document stores suitable for distributed and cloud-based applications.

— **Simplified Development** : The direct mapping of data to JSON-like structures reduces the impedance mismatch between the database and application code.

## 2.3 Challenges

— **Weaker ACID Compliance (Atomicity, Consistency, Isolation, Durability)** : Document stores often prioritize availability and scalability over strict consistency, which may not be ideal for transactional applications.

— **Data Redundancy** : Denormalized data structures can lead to duplication and require additional effort for data integrity management.

— **Limited Query Flexibility** : While document stores are efficient for certain operations, complex joins and relational operations can be challenging or unsupported.

## 2.4 Use Cases

Document stores are particularly well-suited for applications that require flexible data models and high scalability. Common use cases include :

— **Real-time Applications** : Applications such as messaging platforms, collaboration tools, and notification systems benefit from the low latency and schema-less design of document stores.

— **Catalogs and Content Management Systems (CMS)** : E-commerce platforms and CMS systems leverage the flexibility of document stores to manage diverse product information or multimedia content.

— **Web Applications** : Modern web applications use document stores for their ability to handle large volumes of traffic, user-generated content, and dynamic data.

# 3 Firebase Realtime Database

## 3.1 Realtime Database

Real-time Database is a cloud-hosted database which organized collection of data and can be stored locally on your computer or can be stored in cloud storage. Data is stored systematically as *JSON format*, synchronized continuously to each associated client and fetched when required.

Real-time processing or computing is a system which is subject to "real-time constraints". It means that this system is able to "control an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [Mar65]

Real-time database follows several time constraints which can be *hard*, *firm* or *soft* and one must add to them the **temporal consistency** composed by the ***absolute consistency*** and *relative consistency* of the data.

## 3.2 What is Firebase?

Firebase is a cloud-hosted document storage database system developed by Google which, unlike SQL-based databases, doesn't use tables, instead it uses collections of JSON data. NoSQL databases have the specificity of not being relational because they can store data in an unstructured format. It is more scalable and it contains an expression-based rules language, called the Firebase Realtime Database Security Rules, which define how data is structured, and which users have rights to that data.

We use DB security rules to specify who has access to what pieces of data and how DB should be structured. Cloud Storage Security Rules manage the complexity by allowing you to specify path-based permissions. In just a few lines of code, you can write authorization rules that restrict Cloud Storage requests to a certain user or limit the size of an upload. The rules are securely stored in the RTDB on our servers.

Whenever data changes in the database, events are fired in the client code, and you can then handle and update the state of your user interface in response. When updating data in real-time database, the data is stored on cloud then the system will notify interested devices in milliseconds. The figure below 3.1 illustrates how it's done.

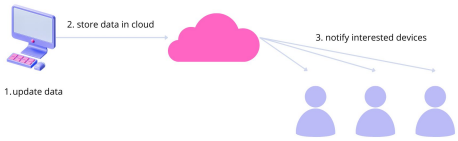


FIGURE 3.1: Firebase Real Time Database - Update Data Architecture

## 3.3 Firebase Features

— With **Realtime DB, file storage and hosting solutions**, backend infrastructure problems are solved. It provides storage facilities. It can store and retrieve content like images, videos and audio directly from client SDK. Uploading and downloading is done in the background. Data stores are safe, and the only authorized user can access it.

— By using the **app indexing** component, you easily get to index your application in Google Search. For instance, if your application is already installed in the user's device when he searches for related content, it will live your app directly from the search results. If users have not installed your application yet, an install card shows up in search results

— **Firebase Authentication** gives backend services to confirm clients over an application. It supports several authentication methods or integrate any existing authentication back ends you own. Users can be allowed to sign in to a Firebase app either by using FirebaseUI [1] as a complete drop-in authentication solution or by using the SDK to manually integrate one or more sign-in techniques [2].

— **Analytics component** is designed only for firebase. It brings insights into how well these components are working for admin and users. This feature enables the application developer to understand how users are using his application. The SDK captures events and properties on its own and allows you to get custom data. The dashboard provides details like your most active user or what feature of your application is used most. It also provides you with summarized data.

## 3.4 Strengths and Limitations

### 3.4.1 Strengths

Firebase Realtime Database offers several strengths :

1. An open-source interface containing loads of authentication solution that can be customized to match the app style. It is the easy solution since it provides a complete sign-in system for the app and it is directly integrated into the system.
2. This way is used when we want to manually add one or several sign-in methods.

— **Schema-less Structure** : Firebase stores data in a schema-less JSON tree, offering flexibility in data modeling. Developers can easily adjust data structures without worrying about migrations or schema changes.

— **Real-time Synchronization** : One of the core features of Firebase is its ability to synchronize data in real-time across connected devices. This makes it ideal for collaborative applications, chat systems, or live dashboards.

— **Ease of Integration** : Firebase integrates easily with other Firebase services, such as Authentication, Cloud Functions, Hosting, and Storage reducing development overhead.

— **Offline Capabilities** : Applications built on Firebase can function offline by caching data locally. Once the device regains connectivity, the database synchronizes changes, ensuring a seamless user experience.

### 3.4.2 Limitations

While Firebase Realtime Database offers significant advantages, it also has limitations that we should consider :

— **Scalability Constraints** : Firebase Realtime Database struggles to handle very large datasets or high write-throughput scenarios efficiently. It may not be ideal for enterprise-scale applications with complex queries.

— **Data Structure Challenges** : Data is stored as a JSON tree, and deeply nested structures can become difficult to manage and query. Developers often need to denormalize data, leading to potential redundancy.

— **Limited Query Capabilities** : Firebase's querying functionality is relatively basic compared to other databases like Firestore or relational databases. Complex queries, joins, and advanced filtering are not supported.

— **Cost** : Firebase's pricing model (pay-as-you-go) can become expensive as the app scales. The operations will be highly operated such as read and write. There will be large files hosted and frequent cloud function invocations.

# 4 Google Cloud Firestore

Cloud Firestore is a scalable NoSQL cloud database developed by Google, designed to support both client-side and server-side development. Built on Google Cloud infrastructure, Firestore provides a robust platform for storing and syncing data in real-time, enabling users to see updates as they happen. With its simplicity, flexibility, and high performance, Firestore has become a key database for modern digital projects.

As a NoSQL database, Firestore moves away from traditional rows and tables, adopting a flexible data model based on Documents, which are grouped into Collections. This approach is similar to a folder containing files. Firestore optimizes data storage for large collections of small documents, allowing developers to easily adapt the hierarchy or reorganize the data model as needed.

Every document in Firestore is uniquely identified by its location in the database, and Firestore offers SDKs tailored for various platforms, including mobile, web, and backend systems.

## 4.1 Firestore Features

— **Real-time Sync** : Firestore uses real-time listeners to monitor changes in the database. When modifications occur, all connected clients receive updates instantly without requiring manual refreshes.

— **Flexible Data Structure** : Firestore's document-based model allows for flexible, hierarchical data storage. Documents store data in JSON format and can include nested objects or arrays where collections group related documents.

— **Automatic Indexing** : Firestore automatically indexes all data fields, enabling efficient

  — Single-field indexes : Automatically created for all fields.

  — Composite indexes : Customizable and used for complex queries involving multiple fields. This automatic indexing eliminates the need for manual optimization and significantly enhances query performance.

— **Offline Mode** : Firestore supports offline data persistence through local caching. Applications can access and modify data while disconnected from the Internet. Once reconnected, Firestore synchronizes local changes with the cloud backend.

— **Conflict Management** : Firestore's design handles distributed environments, where multiple clients may update the same document or collection concurrently, by synchronizing changes and resolving conflicts automatically when possible.

— **Serverless** : Firestore is fully managed by Google Cloud, requiring no server setup, maintenance, or scaling configuration. We interact with the database through APIs, and Firestore dynamically allocates resources to match application demand.

## 4.2 Strengths and Limitations

### 4.2.1 Strengths

Google Cloud Firestore offers several advantages that make it a preferred choice for modern applications :

— **Flexible Data Model** : Firestore organizes data into documents and collections, providing a highly flexible structure for storing hierarchical or semi-structured data simplifing real-world data representation compared to Firebase Realtime Database's flat JSON tree.

— **Powerful Query Capabilities** : Firestore supports more advanced querying options, including compound queries, range filtering, and sorting. These features make it suitable for applications requiring complex data retrieval operations.

— **Automatic Scaling** : Firestore automatically scales horizontally to handle varying workloads, from small to bigger applications. This ensures consistent performance under high traffic or large datasets without doing it manually.

— **Real-time Sync with Offline Support** : Firestore supports real-time data synchronization and allows applications to function even in environments with unavailable connection. Once restored, offline writes are synchronized with the cloud.

— **Serverless Architecture** : Firestore eliminates the need for server management. Developers can focus on building applications, while Firestore automatically handles infrastructure, scaling, and updates.

### 4.2.2 Limitations

Despite its strengths, Google Cloud Firestore has certain limitations :

— **Query Complexity Constraints** : Firestore does not support features like joins, aggregations, or full-text search natively. Using additional tools to implement these capabilities can increase complexity.

— **Pricing and Cost Management** : Firestore follows a pay-per-use model based on read/write operations and storage. High-frequency workloads or poorly optimized queries can lead to significant costs as the application scales.

— **Limited Bulk Operation** : Firestore does not natively support bulk insert, update, or delete operations. We must manage such operations with programs, which can increase development effort for handling large datasets.

— **Query Index Limitations** : While Firestore automatically indexes fields, certain complex queries may require manually creating composite indexes. The system enforces limits on the number of indexes, which may impact very large-scale applications.

— **Latency** : Although Firestore is highly scalable, applications querying massive datasets or handling high levels of multiple reads and writes might experience slight latency, especially under peak loads.

# 5 Benchmarking Firebase Realtime Database and Firestore

## 5.1 Benchmarking

It's the technique of methodically evaluating and calculating a database management system's (DBMS) performance under particular workloads. It involves executing a series of typical queries and actions (i.e. read, write, update, delete...) to evaluate the database's performance in various scenarios. The aim is to understand the database's capability and limitations, especially with regard to speed, scalability, and resource efficiency. **Each query was executed 6 times**.

## 5.2 Why benchmarking?

The goal of benchmarking is answering important concerns regarding a database's performance and suitability for a particular application. In our e-commerce project benchmarking is useful because it :

Checks to see **scalability**; if the database can manage growing volumes or traffic of data without experiencing a noticeable decline in performance. That's why, we're testing with dataset sizes of 1,000 to 25,000 records, which will show us if the database scales exponentially or linearly.

It assesses the database's speed in **response time**, such as adding an order, updating price levels, or retrieving a product by ID. Applications that need real-time performance, like e-commerce platforms, need stable performance.

Examines the **efficiency of resources** & the database's utilization of the CPU and storage for every activity. This is crucial for databases stored in the cloud, because resource usage affects pricing.

## 5.3 Benchmarking Process

The E-commerce data model consists of collections as follows product, user and order. We used an existing dataset. Each collection has a field like the sizes, feedbacks, ...etc.

Here's an example 5.1 of the JSON representation of a section of the product collection :

```
{"product_id":"1",
"name":"Microwave",
"description":"Reliable and affordable.",
"price":36.52,
"category":["Home"],
"rating":2.8,
"reviews":[{"user":"user840","comment":"Limited edition and exclusive design.","rating":4.4},
           {"user":"user713","comment":"Durable and long-lasting.","rating":1.8},
           {"user":"user197","comment":"High-quality product with excellent features.","rating":4.7},
           {"user":"user340","comment":"Durable and long-lasting.","rating":2.3},
           {"user":"user190","comment":"Limited edition and exclusive design.","rating":1},
           {"user":"user954","comment":"Durable and long-lasting.","rating":1.8},
           {"user":"user281","comment":"Top-rated by customers.","rating":1.6},
           {"user":"user538","comment":"Reliable and affordable.","rating":4.8},
           {"user":"user761","comment":"Top-rated by customers.","rating":2.6}],
"images":["image2.jpg","image3.jpg"],
"stock":486,
"date_added":"2024-09-04T13:49:57.586Z"},
```


FIGURE 5.1: Representation of Product Collection

## 5.4 Benchmarking Results

To evaluate the performance of Firesbase Firestore, we benchmarked it against PostgreSQL, a traditional relational database. Identical queries were designed for both systems to ensure consistency in comparison, focusing on key database operations such as retrieval, updates, deletions, and more. We worked on showing why Firebase Firestore is the perfect choice for our application.

### 5.4.1 Retrieve & Update Queries

PostgreSQL outperforms Firestore in retrieving all records, especially when the size of the dataset size increases because it is optimized for executing bulk data retrieval using efficient query execution plans and indexing. While Firestore's document-based structure is instead designed for retrieving specific documents or subsets of data *on-demand*. The performance is exponential as dataset size increases, reflecting its limitations for bulk operations. But its real-time synchronization capabilities and automatic data propagation make it a better fit for applications that prioritize real-time updates to subsets of data rather than retrieving large data.

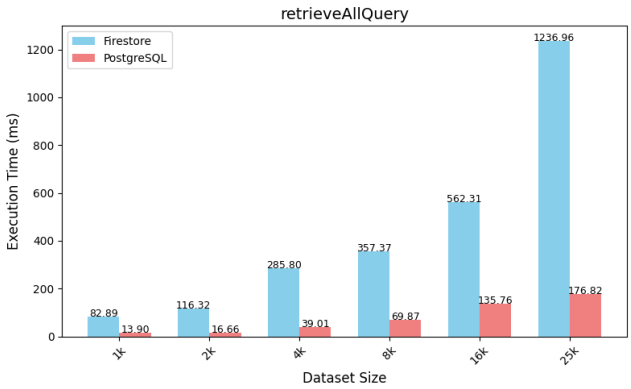


FIGURE 5.2: Retrieve All

In retrieving single records, PostgreSQL demonstrates its strengths with extremely low query times due to its relational nature, optimized indexing, and efficient use of primary keys. Firestore, while slower, maintains relatively consistent performance as dataset sizes grow. This linear behavior shows that Firestore scales predictably for targeted data queries. With proper structuring and exploitation of Firestore's real-time listeners and caching mechanisms, single-record retrievals can

be significantly optimized for real-time applications, making Firestore an ideal choice for use cases like apps where individual data updates are more critical than bulk operations.

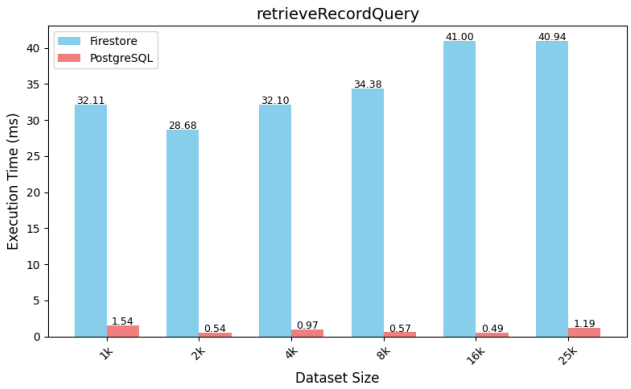


FIGURE 5.3: Retrieve Record Query

PostgreSQL again offer smaller query times due to its ability to efficiently process transactions and optimize query executions. But Firestore supports distributed, concurrent updates with built-in conflict resolution, which ensures data consistency in real-time environments. While PostgreSQL might be a better choice for applications requiring high-frequency batch updates, Firestore's linear growth in update query times and its ability to synchronize updates across devices without manual interventions make it the right choice for real-time, user-centric applications where immediate reflection of changes is critical.

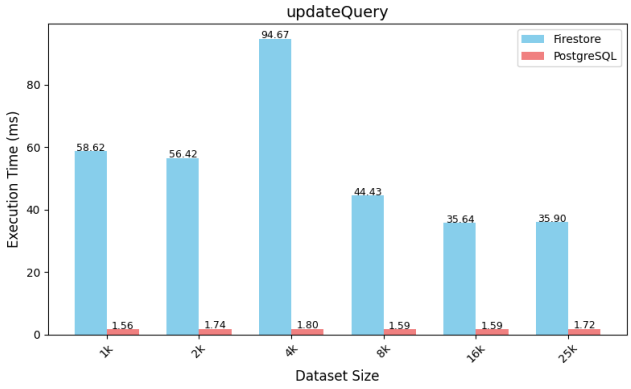


FIGURE 5.4: Update Query

### 5.4.2 Add & Delete Queries

For delete queries, PostgreSQL shows better performance across all dataset sizes due to its indexing mechanisms and transaction handling, which are optimized for structured data storage. Firestore, in contrast, demonstrates a linear growth indicating that it handles increasing workloads predictably. While PostgreSQL's speed is advantageous for applications with strict low-latency requirements, Firestore's real-time data synchronization and horizontal scalability make it more suitable for dynamic distributed environments.


FIGURE 5.5: Delete Query

Similar trends are observed in add query performance, where PostgreSQL again outperforms Firestore demonstrating consistent and reliable performance across benchmarks. But Firestore's ability to manage semi-structured data without requiring rigid schemas provides a stronger advantage for modern applications with evolving data needs. And, its linear scalability ensures that as data volume grows, performance remains predictable. Its integration with real-time systems allows near instantaneous data visibility for users even if it means a slower query speeds.

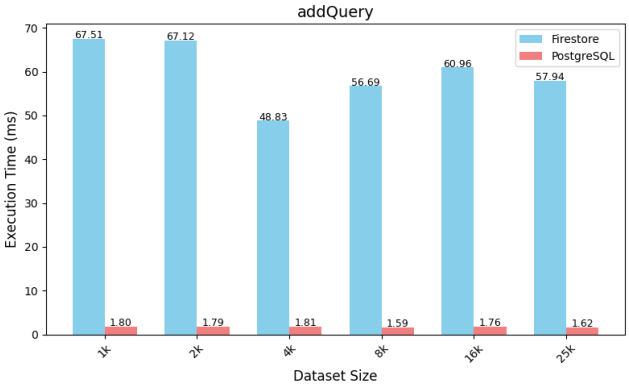


FIGURE 5.6: Add Query

### 5.4.3 Compound Query

When analyzing the performance of compound queries, PostgreSQL shows a significant advantage in terms of execution speed because of the way it optimizes query, using multi-column indexing, joins, and query execution plans. Firestore struggles as the size growth showing an exponential performance because it processes compound queries in a way that requires multiple document reads, which becomes more and more costly for larger datasets. Firestore can still be the better choice when compound queries are designed to exploit its indexing strengths and query filters.

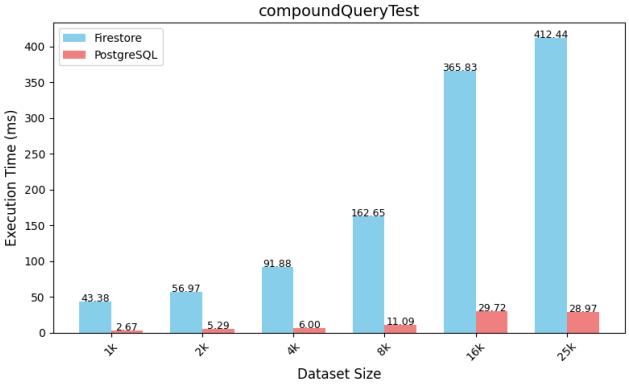


FIGURE 5.7: Compound Query

### 5.4.4 Paginated Query

For paginated queries, PostgreSQL again shows better results due to its ability to execute offset and limit-based pagination with minimal overhead while Firestore shows slower but quite stable performance, reflecting its ability to handle pagination through cursors rather than offsets. It avoids scanning large amounts of unnecessary data but the cost increases proportionally because each query still requires document reads, even if only a subset of data is retrieved. However, by structuring data to minimize document reads and optimizing queries to support cursors, Firestore can meet the needs of applications requiring efficient navigation through paginated data.

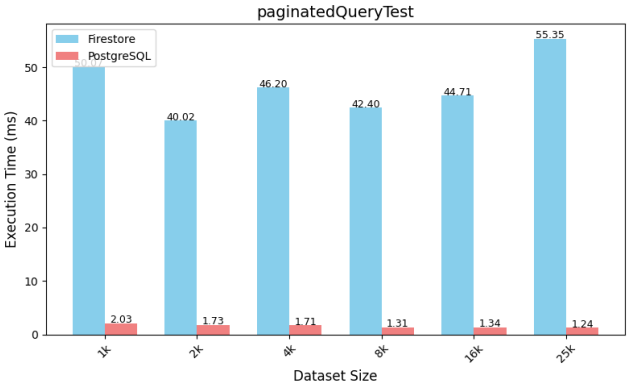


FIGURE 5.8: Paginated Query

# 6 Application

## 6.1 Description

Our application is a simple e-commerce platform designed to demonstrate the integration of Firebase and Firestore and the first goal of the application is to show how they work together to provide an efficient and scalable way for handling real-time product updates, searching, filtering, and sorting.

The application includes product listings with several information. Users can search for products by name, filter them based on categories, price or ratings and sort them by price or newest listings. The real-time synchronization capabilities of Firebase ensure that any changes made to the database are reflected in real time across all users' devices.

## 6.2 Database Design

The database is structured using Firestore's NoSQL document model. Each product is stored as a document within a Firestore collection, and each document contains fields for the product's name and other relevant information. Some key design decisions for the database include :

— **Product Collection** : All products are stored in a single collection, with each product stored as a document within that collection. Each document contains fields for the product's name, price, description, rating, category, and the date the product was added.

— **Indexes** : To optimize performance, indexes are created for frequently queried fields, such as "price", "rating", and "date_added". These indexes ensure that queries on these fields are fast and efficient.

## 6.3 Integration of Firestore/Firebase

Firebase and Firestore are integrated in the application such as Firebase provides tools for enabling real-time updates, while Firestore being the main data storage for product information. Firebase's 'onSnapshot' listener is used to provide real-time updates to product listings. Whenever a change occurs in the database, the application automatically reflects the change without needing to refresh the page. Users can search for products by name (which supports partial matching)

but also filter the product list and there are applied directly to Firestore queries. The application allows users to sort products by price (ascending or descending) or by the date they were added. Firestore's "orderBy" function is used to apply sorting dynamically.

### 6.3.1 How Firestore/Firebase Enhances Performance

Firestore and Firebase enhance the performance of the application in several ways :

— **Real-Time Synchronization** : Firestore's real-time synchronization ensures that the application's data is always up-to-date. When a change occurs, all connected clients immediately receive the update without needing to refresh the page.

— **Efficient Querying** : Firestore supports powerful querying capabilities which allows for efficient retrieval of data. With indexes, even complex queries are executed quickly, even when dealing with large datasets.

— **Scalability** : Firestore's NoSQL model is designed for scalability, allowing the application to easily scale as the number of products grows. This flexibility allows for easy updates to the database structure without disrupting the application's performance.

— **Offline Support** : Firestore includes built-in support for offline data access, our application will continue to work even when the user is not connected to the internet. Data is automatically synchronized when the connection is restored.

— **Integration with Firebase** : Firebase provides several services, such as Firebase Hosting for fast delivery of static content, which integrate with Firestore simplifying the implementation. This makes it easy to build an application with a minimal setup.

### 6.3.2 Challenges Faced During Implementation

During the development of the application, several challenges were encountered :

— **Handling Large Data Sets** : As the number of products increased, retrieving all products at once became inefficient. To address this, pagination was implemented, allowing the application to load products in chunks and only fetch additional data according to the need of the user.

— **Managing Real-Time Listeners** : Firebase's real-time listeners are a powerful tool, but managing them can be tricky. For example, when a user switches between filters or search criteria, it's important to unsubscribe from the old listeners to avoid memory leaks.

— **Query Limitations** : Firestore's querying model has some limitations compared to traditional SQL databases. For example, Firestore does not support complex joins or subqueries, which required us to change the data model and how related data is retrieved.

— **Cost for High-Scale Applications** : Firestore's cost model, which is based on document reads, writes, and storage, can become expensive for applications with high traffic or frequent updates. Real-time listeners exacerbate this issue, as they continuously consume read operations.

# 7 Conclusion

Firebase Firestore offers several key advantages for applications requiring real-time synchronization, scalability, and flexibility. Its ability to provide real-time updates ensures that users see changes immediately, without refreshing the page. Combined with Firestore's scalability and ability to handle semi-structured data, makes it an ideal choice for dynamic applications like e-commerce.

When choosing between Firestore and relational databases, it is clear that Firestore should be prioritized for applications that need real-time and flexible data models. But we still need to take into account Firestore's pricing model. Based on document reads, writes, and storage, it can become expensive for high-traffic applications or frequent updates. As the application scales and the number of reads and writes increases, the cost can rise significantly, making it crucial to optimize queries and structure data efficiently. These limitations must be considered when planning for larger applications

While PostgreSQL excels in large data retrieval and complex queries, Firestore's performance remains stable for specific queries, ensuring efficient handling of real-time data in environments where scalability is essential. PostgreSQL remains a better choice for applications with complex relationships, requiring ACID compliance and systems that need high-efficiency data analysis or transactional integrity.

# Bibliography

[Elb21] ELBADAWY, Aymen (2021). « Data Model ». In : URL : https://aymanelbadawy.com/data-model/.

[Nat18] NATH, Asoke (2018). « Real-Time Communication Application Based on Android Using Google Firebase ». In : URL : https://www.researchgate.net/profile/Asoke-Nath-4/publication/324840628_Real-time_Communication_Application_Based_on_Android_Using_Google_Firebase/links/5ae721760f7e9b9793c82cbf/Real-time-Communication-Application-Based-on-Android-Using-Google-Firebase.pdf.

[Res20] RESEARCH, TechVision (11 February 2020). « Graph Databases, GraphQL and IAM ». In.