What is Firebase?
Firebase is a platform developed by Google for creating mobile and web applications. It provides various services and tools that help developers build, improve, and grow their apps easily and efficiently.

Examples of Firebase
1. Real-time Database: A database to store and sync data in real-time.
2. Authentication: Services to authenticate users using various methods like email, Google, Facebook, etc.
3. Cloud Functions: Run backend code in response to events triggered by Firebase features and HTTPS requests.
4. Hosting: Provides fast and secure hosting for web apps.
5. Cloud Firestore: A scalable database for storing, syncing, and querying data for mobile and web apps.
6. Cloud Storage: Object storage for files, images, etc.

Main Parts of Firebase
1. Develop: Offers tools like Firestore, Authentication, Real-time Database, and Cloud Functions to help developers build apps.
2. Quality: Provides Crashlytics, Performance Monitoring, and Test Lab to maintain the app quality.
3. Analytics: Gathers user and event data to help in making informed decisions on app improvement.
4. Grow: Includes services like In-App Messaging, Notifications, and Remote Config to help grow the user base and engagement.

Use Cases of Firebase
1. Real-time Chat Apps: Using Real-time Database and Authentication for instant message delivery and user authentication.
2. E-Commerce Apps: Utilizing Firestore for product catalog management and Authentication for user sign-in.
3. Analytical Dashboard: Utilizing Analytics to collect user data and behaviors to make informed decisions.
4. Serverless Applications: Using Cloud Functions to run server-side code without managing servers.

Why Do We Need to Use Firebase?
Firebase allows developers to build applications quickly with less effort because it provides a range of services and tools that handle various development aspects such as database management, authentication, and analytics. It is especially beneficial for developers without extensive backend knowledge as it reduces the need for server management and backend coding.

Why We Should Not Use Firebase?

1. Cost: Firebase can become expensive as the app scales due to its pricing model.
2. Limited Querying: The databases in Firebase have some limitations regarding complex queries.
3. Lock-In: Being a proprietary service, migrating to another platform can be challenging.

Relation between Firebase and Firestore

Cloud Firestore is a part of Firebase. It is a NoSQL database service that allows developers to store, sync, and query app data easily. Firestore provides real-time synchronization and robust querying, making it suitable for various kinds of app development projects within the Firebase ecosystem.

Key Basics About Firebase

1. Services
   a. Authentication: Manages user identity.
   b. Firestore: Offers NoSQL database.
   c. Real-time Database: Provides real-time data syncing.
   d. Cloud Functions: Executes backend code in response to events.
   e. Hosting: Serves web applications.
   f. Storage: Stores files and media.
   g. Analytics: Analyzes user behavior and usage.
   h. Crashlytics: Monitors app performance and crashes.

2. Platform
   a. Web: Supports web app development.
   b. iOS/Android: Provides libraries and SDKs for mobile development.

3. Development Cycle
   a. Build: Use Firebase services to create apps.
   b. Improve: Utilize quality and analytics tools to enhance app performance and user experience.
   c. Grow: Deploy growth features to expand the user base.

What is Firestore?

Firestore, also known as Cloud Firestore, is a flexible, scalable NoSQL cloud database from Firebase that allows you to store and sync data for your mobile and web applications in real-time. With Firestore, you can store your app's data in documents organized into collections, and it offers seamless synchronization, offline access, and robust querying capabilities.

Characteristics of Firestore:

1. Document-Oriented: Data is stored in documents, which are organized into collections.

2. Real-time Synchronization: Changes made to the data get automatically updated on connected client devices in real-time.
3. Offline Support: Firestore SDKs cache data locally, allowing apps to function offline. Once back online, the local data syncs with the cloud.
4. Deep Queries: It supports complex querying capabilities without the need for secondary indexing.
5. Scalable: Built on top of Google Cloud Platform, Firestore automatically scales with the demands of your app.
6. Multi-Region Replication: Data is automatically replicated across multiple regions, ensuring high availability and resilience.
7. Security: Provides robust security rules to safeguard data access and manipulation.

How Firestore Scales Data and Database?
Firestore's scalability is rooted in its data model and infrastructure:

1. Collections and Documents: The data model is composed of collections (akin to tables) and documents (akin to records). This allows for flexible, hierarchical organization.

2. Sharding: As you add more data, Firestore distributes your data across multiple servers to balance the load and ensure consistent performance. The system decides how to split the data based on the amount of read and write activity, and it automatically handles the distribution.

3. Multi-Region Replication: Firestore automatically replicates your data across multiple data centers in different regions. This not only provides redundancy but also ensures that the database remains performant even when it's serving global users.

4. Horizontal Scaling: Instead of making a single server more powerful (vertical scaling), Firestore adds more servers (machines) to the system (horizontal scaling). This means as your app grows and demands more resources, Firestore can easily accommodate this growth by adding more servers to handle the traffic.

To simplify: Imagine Firestore as a massive library. As you get more books (data), instead of getting a bigger shelf (server), Firestore opens new library branches (servers) in various locations. This ensures everyone can access their favorite book (data) quickly and without waiting in a long queue, no matter how many people (users) are reading.

Example of Firestore Application or DB Name
- DB Name: MyChatAppDB

- Application: A real-time chat application where users can send and receive messages instantly.

How Do We Store and Retrieve Data in Firestore?

Store Data:
```javascript
// Reference to the Firestore database
const db = firebase.firestore();

// Add a new document to the 'users' collection with ID 'user123'
db.collection('users').doc('user123').set({
    name: 'Alice',
    age: 30,
    email: 'alice@example.com'
});
```

Retrieve Data:
```javascript
// Get a document from the 'users' collection with ID 'user123'
db.collection('users').doc('user123').get().then((doc) => {
    if (doc.exists) {
        console.log("User Data:", doc.data());
    } else {
        console.log("No such document!");
    }
}).catch((error) => {
    console.log("Error getting document:", error);
});
```

Benefits of Schema-less Database & Use Case

Benefits:
1. Flexibility: You can add or modify fields without having to alter the whole database schema.
2. Quick Iteration: Enables faster development, especially in the early stages when data requirements might change frequently.
3. Scalability: Can handle large volumes of structured, semi-structured, or unstructured data.

Use Case: A startup building a social media application may not have a fixed structure for user profiles initially. They might start with basic fields like 'name' and 'email'. Later, they could decide to add 'hobbies', 'workplace', or 'education' without restructuring the entire database.

Firestore - Native and Datastore Mode

Firestore originally evolved from Google Cloud Datastore. The distinction is mostly historical:

1. Native Mode: This is the typical Firestore that you use with Firebase apps. It's optimized for serverless setups with Firebase services.
   - Example: Building a mobile chat app where Firestore (in native mode) manages real-time messaging.

2. Datastore Mode: This mode is optimized for Google Cloud-based apps and has its roots in the older Google Cloud Datastore product.
   - Example: A GCP web service that manages data using the Datastore mode of Firestore.

However, Google has been moving toward unifying the experience, and developers are encouraged to use Firestore in Native mode for new projects.

Types of Firestores

Firestore is a single product; there aren't multiple "types" of Firestore. But if you are referring to the database's modes, then as mentioned, there's "Native" mode and "Datastore" mode.

CRUD Process in Firebase

CRUD stands for Create, Read, Update, Delete. Here's a simple example in Firestore:

1. Create:
```javascript
db.collection('users').add({
    name: 'Bob',
    age: 25,
    email: 'bob@example.com'
});
```

2. Read:
```javascript
db.collection('users').doc('user123').get().then((doc) => {
    console.log(doc.data());
});
```

3. Update:
```javascript
db.collection('users').doc('user123').update({
    age: 26
});
```

```

4. Delete:
```javascript
db.collection('users').doc('user123').delete();
```

This is a very high-level overview. In real applications, there are considerations for error handling, handling collections, streaming data updates, and more.

Limitations of Firestore:

1. Cost: As the number of reads, writes, and stored data increases, costs can escalate.
   - Example: A high traffic app can rack up costs due to a large number of reads and writes.

2. Complex Queries: Firestore doesn't support SQL-like joins or complex queries.
   - Example: You can't easily fetch all books written by authors who live in a certain city without structuring your data very specifically or making multiple requests.

3. Limited Transactions: Transactions are confined to a set limit of 500 writes.
   - Example: If you're updating stock for 600 items in a single transaction, it won't work.

4. Cold Starts: Occasionally, there can be latency spikes on the first request to Firestore after a period of inactivity.
   - Example: An app's first request of the day might experience a slight delay.

Native Mode vs. Datastore Mode:

1. Native Mode:
   - Tailored for Firebase and mobile/web development.
   - Integrated with Firebase features like Cloud Messaging, Authentication, etc.
   - Real-time updates and offline access are key strengths.

2. Datastore Mode:
   - Older mode, primarily for Google Cloud Platform apps.
   - Lacks real-time capabilities of Native mode.
   - Integrated more deeply with GCP's older set of tools and services.

Over time, Google has been encouraging developers to use Native mode due to its more modern capabilities and integrations.
```

Firestore API:

The Firestore API allows developers to interact with their Firestore database, performing operations like reading, writing, updating, and deleting data.

How It Works:
1. Initialization: Set up a connection to Firestore using Firebase SDK. This involves initializing the Firebase app with configuration details.

2. Request & Response: Make a request to the Firestore database (e.g., retrieve a document, update data). Firestore processes the request and sends back a response.

3. Real-time Listeners: Unlike traditional databases, Firestore allows you to set up listeners on your data. When the data changes, Firestore pushes the updated data to the app without needing a new request.

Process:
1. Setup: Install Firebase SDK, initialize your app, and get a reference to the Firestore database.

2. Interact with Data: Use the methods provided by the SDK to make CRUD operations:
   - `db.collection('...').add({...})` to create.
   - `db.collection('...').doc('...').get()` to read.
   - `db.collection('...').doc('...').update({...})` to update.
   - `db.collection('...').doc('...').delete()` to delete.

3. Listeners: To get real-time updates:
   - `db.collection('...').onSnapshot(snapshot => {...})`

4. Security: Implement security rules to control access and modifications to the data.

Examples:
- Write Data: Store a new book in the "books" collection:
```javascript
const db = firebase.firestore();
db.collection('books').add({
    title: 'The Great Gatsby',
    author: 'F. Scott Fitzgerald'
});
```

- Read Data: Fetch details of the book with ID 'book123':
```javascript
db.collection('books').doc('book123').get().then(doc => {
```

```
    if (doc.exists) {
        console.log(doc.data());
    }
});
```

Remember, the Firestore API is vast, providing a plethora of functionalities ranging from basic CRUD to advanced querying, batch operations, and more.

Example Scenario:

Background:
You're working on a mobile application for a book club named "Readers' Haven". Members of the club can read, review, and recommend books to each other. As the developer, you chose Firestore as the backend database for this app.

Firestore Structure:
You have two main collections:
1. `books`: Each document in this collection represents a book with fields like `title`, `author`, `genre`, and `summary`.
2. `reviews`: Each document here is a review written by a club member with fields like `bookId` (referring to a book in the `books` collection), `userId`, `rating`, and `comment`.

Problem Statement:
A feature request comes in: Club members want a "Recommended Reads" section in the app that showcases the top 3 books with the highest average rating, along with their latest review. However, the catch is that only books with more than 5 reviews should be considered. How would you fetch this data from Firestore?

Solution:

Step 1: Fetch books with more than 5 reviews:
You can't directly perform complex aggregations in Firestore like you would in SQL databases. So, you'd have to maintain a `reviewCount` field in each `books` document, incrementing it every time a review is added.

Step 2: Calculate average ratings:
Similarly, you might maintain an `averageRating` and `totalRating` field in each `books` document. Every time a review is added, you update both `totalRating` and then compute the `averageRating` as `totalRating/reviewCount`.

Step 3: Fetch the top 3 books:
With the above fields in place, you can fetch the top 3 books based on `averageRating`:

```javascript
const booksRef = firebase.firestore().collection('books');
booksRef.where('reviewCount', '>', 5)
        .orderBy('averageRating', 'desc')
        .limit(3)
        .get()
        .then(snapshot => {
            let topBooks = [];
            snapshot.forEach(doc => {
                topBooks.push(doc.data());
            });
            // Now you have the top 3 books
        });
```

Step 4: Fetch the latest review for each of these books:
For each book from the result, fetch the latest review from the `reviews` collection:

```javascript
const reviewsRef = firebase.firestore().collection('reviews');
topBooks.forEach(book => {
    reviewsRef.where('bookId', '==', book.id)
              .orderBy('timestamp', 'desc') // Assuming you have a timestamp field for each review.
              .limit(1)
              .get()
              .then(snapshot => {
                  let latestReview;
                  snapshot.forEach(doc => {
                      latestReview = doc.data();
                  });
                  book.latestReview = latestReview;
              });
});
```

Optimization Note:
To optimize and reduce the number of reads, you might consider denormalizing your data. This means storing redundant data in multiple places to optimize read-heavy operations. For example, the latest review for a book can be stored inside the `books` document itself. However, always be mindful of Firestore's document size limits and costs associated with multiple writes.

Explanation:

- Firebase: Firebase is a Backend-as-a-Service (BaaS) platform. It's like a big toolkit for developers. Inside this toolkit, you have multiple tools/services like Firestore (database), Firebase Authentication (for user management), Firebase Hosting (for web hosting), Firebase Cloud Messaging (for push notifications), and many more. For instance, if you're building a house (app), Firebase provides you with all the essential tools you'd need.
- Firestore (Cloud Firestore): Firestore is just one of those tools in the Firebase toolkit. Specifically, it's a real-time NoSQL cloud database. It's designed to be super scalable and is optimized for building mobile and web applications. Using our house (app) analogy, if Firebase gave you all the tools, Firestore would be the foundation or the main structure, where all your data (furniture, belongings) resides.

In essence, Firestore is a part of Firebase. While Firebase encompasses a broader range of services and tools, Firestore is specifically focused on being a cloud-based NoSQL database.

Aspect	Firebase	Firestore (Cloud Firestore)
What is it?	A platform that provides various tools and infrastructure needed to build apps.	A NoSQL cloud database within Firebase that lets you store and sync data between your apps and the cloud in real-time.
Nature	Suite of services.	One specific service within Firebase.
Primary Role	Offers a range of tools and services from hosting to machine learning.	Specifically designed for storing, querying, and syncing data.
Example	Think of Firebase as a toolbox where each tool serves a different purpose (hosting, database, authentication).	Think of Firestore as one specific tool in that toolbox - a screwdriver, perhaps, essential for a certain set of tasks (handling data).