Effective Database Structures for Session-Based Realtime Databases

CS848 Project – Karthik Prasad

I. Introduction

This project will be evaluating the effectiveness of different database structures for session-based realtime databases. It will be focusing on session-based data that is frequently used to monitor user activity in the application of realtime databases which asynchronously update all users instantaneously upon any change to the database. The project will be evaluating two main structures (nested and flat) for what is better in speed and bandwidth usage, as well as seeking to find an optimal solution that may outperform the two conventional structures.

II. Background

It's important to discuss the type and reasoning of both the data and databases chosen for this project. There are a multitude of different data paradigms, but session-based data is something that is becoming increasingly abundant in the modern age. In the past 5 years, there has been a resurgence in the creation of mobile and web supplementary apps to support core products. This is due to companies using data analytics to improve the performance of their products and systems. They are interested in seeing the usage and behavior of their users on their products, which involves tracking all their uses and micro-movements for everything they do in the app. Accordingly, session-based data is a common paradigm that is used to store all this information, and is something used by all the major applications like Amazon, Instagram and Facebook. To use Amazon as an example, they track not only your page visits and mouse clicks, but even where your mouse hovers while you browse different items. This allows them to increase the power of their targeted advertising and increase their product sales, in turn, increasing their revenue.

This type of data can be stored in almost any type of database, so for the purposes of this project, realtime databases were chosen. The distinction of realtime databases is that they allow for asynchronous access to all users of their application through constant updates to everyone whenever any change occurs for any user of the database. Common examples for realtime databases include Supabase and Google's Firebase, which was used for this project due to the free Spark plan it offers. In these databases, the data is commonly stored in an inverted-tree JSON format, which implies that all the data is nested inside multiple layers. Figure 1 below shows how the structure of this database looks using a sample from Firebase's Realtime Database used in the project.

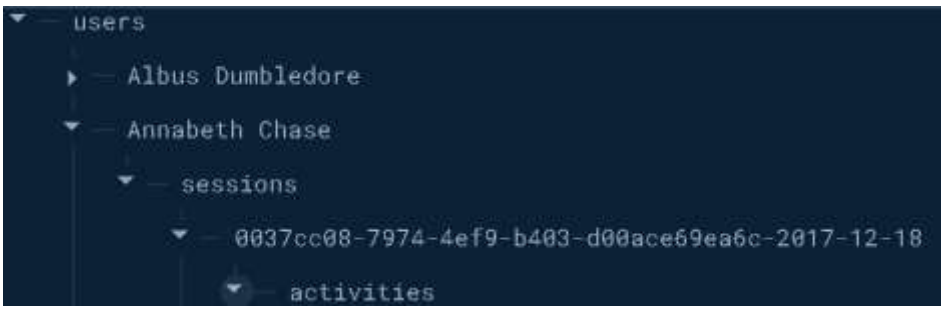

Figure 1. Sample structure of Firebase Realtime Database

There are several common components that one would see in most of these database structures, including the one in Figure 1. They would all include users and their corresponding details like email, age and phone number. They would also usually have sessions that store when users log in and out of the application with the date and time stamps. The sessions would consist of activities that users perform and all the details comprising in that such as activity type, activity duration, activity score, etc. Each of the activities would also contain some state-based or context-based information stored within them that depicts what the user did and how they performed in the activity. There are two common approaches to structuring data in the realtime databases that are shown below:

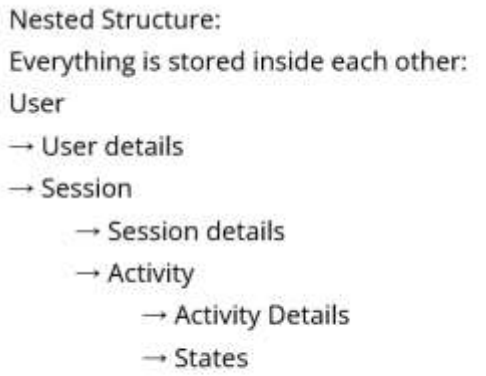

Figure 2a. Nested database structure

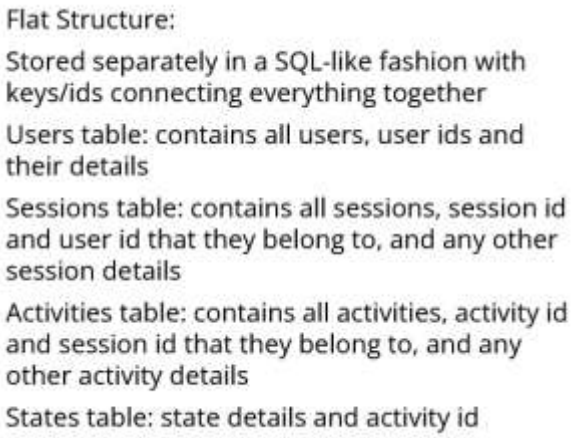

Figure 2b. Flat database structure

Figure 2a shows the conventional nested database structure that uses multiple layers to store everything inside each other. This structure takes after the JSON nested dictionary format and is very useful for user-based information. The nested structure starts with a users section that contains a list of sessions and other user details. Each session would contain a list of activities that were performed in each session. In each activity would be all the details for that activity such as states or other information. On the other hand, figure 2b shows the opposite approach of separating all the data in a SQL-like manner with different tables that are all connected using ids as foreign keys. In this structure, there is a different table for users, sessions, activities and states ids connecting everything together in a linked-list format. The users connect to sessions, which connect to activities, which connect to states.

III. Personal Project Connection

The reasoning behind choosing this project is that it is directly related to what I am working on in my thesis project and I hope to use the findings from the project to improve my thesis project. My thesis is tied to a startup company I am working on called Pragmatica. It is a Virtual Reality (VR) solution to supplement speech and language therapy. In it, our core product is a VR app on the Meta Quest Store with lots of users going through various activities to practice their communication skills. Our users log in to sessions and perform several activities in them. In the activities, they achieve different scores out of 100 based on their performance on how many target states they hit or miss in the activity. Hence, each activity has a large number of states that record every action performed by the user.

In the backend, we use the Firebase Realtime Database to store all the metrics and data associated with our clients and their app usage. In fact, all of this data is immediately uploaded as soon as it is created and can be seen by any clinicians wanting to check the performance and progress of their clients. Thus, there is a data dashboard react web app that enables clinicians to monitor the activities of their users. This data dashboard is also connected to the Firebase Realtime Database, allowing clinicians and us to asynchronously receive updates on our users' performances in the VR app. Therefore, the basis for the project was to achieve the optimal efficiency of our database for both the core VR application and the data dashboard by identifying what database structure would be the best one to use.

IV. Project Goals

Regarding the project basis stated above, multiple goals were determined and outlined below:

1. Test how the nested and flat database structures perform for session-based sample data similar to what is used in Pragmatica's database. This will be performed by assessing their speed and bandwidth usage.
2. If possible, create an optimal solution that takes advantage of the strengths of both database structures.

V. Experiment Setup

To evaluate the nested and flat database structures based on speed and bandwidth usage, the following experiment was set up. A separate Firebase project was created for the "nested" and "flat" structures respectively. Each project was populated with sample data that represented the real data used in Pragmatica's database. The data also consisted of the same users, sessions, activities and states schema that was previously mentioned above. This is because it is not only similar and applicable to my thesis project but also commonly used in SaaS applications with session-based data for data analytics. To assess each of the structures, the firebase database CLI profiler was used to measure each of the databases on their speed and bandwidth usage. For the actual tests, common queries related to the application usage were tested for in both of the databases. These queries were formulated by taking the most common queries used in the data dashboard web application, as these would be the main calls to the database in the real world. The 8 common queries are below:

- User's # of sessions per week, month, year
- User's time spent on app per week, month, year
- User's score progression over time per activity
- User's time spent progression over time per activity
- All users aggregate activity scores per activity
- All users aggregate time spent per activity
- All users aggregate time spent per session
- All users aggregate time spent per week, month, year

For each of the queries, each database structure handles it in different approaches. The nested database approach is quite simple, as it has a different API route that gets called for each user. The flat database structure, however, has different API calls based on which of the users, sessions, activities and states tables are needed for each query. Table 1 depicts what API calls are needed for which queries.

Query	API Call
User's # of sessions per week, month, year	Users, Sessions
User's time spent on app per week, month, year	Users, Sessions
User's score progression over time per activity	Users, Activities, States
User's time spent progression over time per activity	Users, Activities, States
All users aggregate activity scores per activity	Activities, States
All users aggregate time spent per activity	Activities
All users aggregate time spent per session	Sessions
All users aggregate time spent per week, month, year	Activities

Table 1. API calls required for each query in flat database structure

VI. Results

The nested database structure was tested first for its bandwidth usage and speed. Each query was tested 5 times in the Firebase CLI Profiler and the average was taken to obtain consistent measurements.

Path	Total Downloaded (MB)	Average Downloaded (MB)
/users/Percy Jackson	8.51	8.51
/users/Annabeth Chase	8.46	8.46
/users/Grover Underwood	9.17	9.17
/users/Jason Grace	8.64	8.64
/users/Piper McLean	8.08	8.08
/users/Leo Valdez	7.89	7.89
/users/Nico Di Angelo	8.94	8.94
/users/Thalia Grace	8.82	8.82
/users/Hazel Levesque	8.54	8.54
/users/Frank Zhang	8.06	8.06
/users/Harry Potter	8.51	8.51
/users/Hermione Granger	8.36	8.36
/users/Ron Weasley	8.03	8.03
/users/Ginny Weasley	8.8	8.8
/users/Albus Dumbledore	8.09	8.09
/users/Draco Malfoy	8.08	8.08
/users/Severus Snape	7.96	7.96
/users/Tom Riddle	8.46	8.46
/users/Sirius Black	8.34	8.34
/users/Cedric Diggory	7.74	7.74

Figure 3. Nested Database Bandwidth Usage Results

Figure 3 above shows the results displayed by the CLI profiler for the bandwidth usage of the various API calls to 20 different users. The API calls were very consistent between users as they had a very short range of 7.74MB to 9.16MB with an average of 8.37MB.

Path	Avg Execution Speed (ms)
/users/Percy Jackson	292
/users/Annabeth Chase	308
/users/Grover Underwood	420
/users/Jason Grace	394
/users/Piper McLean	309
/users/Leo Valdez	329
/users/Nico Di Angelo	389
/users/Thalia Grace	349
/users/Hazel Levesque	382
/users/Frank Zhang	271
/users/Harry Potter	402
/users/Hermione Granger	448
/users/Ron Weasley	353
/users/Ginny Weasley	367
/users/Albus Dumbledore	336
/users/Draco Malfoy	322
/users/Severus Snape	314
/users/Tom Riddle	381
/users/Sirius Black	314
/users/Cedric Diggory	318

Figure 4. Nested Database Speed Results

Figure 4 shows the average execution speed for each of the users' API calls in the CLI profiler. The speeds ranged from 292ms-448ms with an average of 350ms. Once again, each of the users had comparable execution speeds. This is mainly due to the similarity of the data that they contained.

Path	Total Downloaded (MB)	Average Downloaded (MB)
/states	147.7	147.7
/activities	79.04	79.04
/sessions	7.68	7.68
/users	0.00256	0.00256

Figure 5. Flat Database Bandwidth Usage Results

Figure 5 above shows the CLI profiler's results for the bandwidth usage of the flat database structure. The activities and especially the states tables have most of the data inside of them while the sessions has a little bit and users have almost none in comparison. The total size of the database is also larger due to the addition of ids needed to connect everything.

Path	Avg Execution Speed (ms)
/states	218
/activities	82
/sessions	10
/users	0

Figure 6. Flat Database Speed Results

Figure 6 indicates the average execution speed for querying each of the tables. It trends similarly to the bandwidth usage with the states and activities tables taking up a lot more time than the sessions and users table.

When comparing the bandwidth usage of the two database structures, the nested database performs better than flat for user-based queries as it required ~8MB of downloaded data while the flat structure requires 80-235 MB of downloaded data. On the other hand, the flat database performs better in aggregated queries (all users) as it uses the same 80-235 MB of downloaded data while the nested structure requires at least 167.48 MB to download data for all the users. Now looking at the speed, the nested database structure requires about 35ms for each user. Thus, the nested structure is similar or slightly faster for user based queries, but way slower for aggregate queries, as it would require 699ms to go through the data of all the users. In contrast, the flat database execution speed is drastically faster for aggregated queries, while being slightly slower than the nested database for all user based queries, especially when going through the states (218ms) and activities (82ms).

Overall, neither of the database structures is objectively better than the other, as each database structure has its own pros and cons. The nested structure performs better in user-based queries as it doesn't have to go through data of other users. On the other hand, the flat structure performs better in aggregate queries as it can select the specific data from each user without taking everything from the user (particularly great when not requiring states data). Next, the two structures were evaluated for each query. Table 2 below shows that the flat database performed better in 6/8 of the common queries, making it better for 75% of the scenarios compared to the nested structure.

Query	API Call
User's # of sessions per week, month, year	Flat
User's time spent on app per week, month, year	Flat
User's score progression over time per activity	Nested
User's time spent progression over time per activity	Nested
All users aggregate activity scores per activity	Flat
All users aggregate time spent per activity	Flat
All users aggregate time spent per session	Flat
All users aggregate time spent per week, month, year	Flat

Table 2. Nested and flat database comparison for each query

VII. Creation of the Hybrid Database Structure

Regarding the second goal of the project, a third hybrid approach was created that attempted to take the best of both structures. The flat database did better than the nested database, but it split everything up too much. In the hybrid approach, instead of splitting everything up separately, the nested structure was decoupled into two sections. The first section is users, which contains the user details, session information and a session ID. The second section is activities, which contains activity details and state details, as well as a session ID to connect it to the users section. In order to effectively test this new approach, this hybrid database was also tested on the same 8 common queries as the nested and flat databases. Table 3 below states the API calls needed for each of the queries.

Query	API Call
User's # of sessions per week, month, year	Users
User's time spent on app per week, month, year	Users
User's score progression over time per activity	Users, Activities
User's time spent progression over time per activity	Users, Activities
All users aggregate activity scores per activity	Activities
All users aggregate time spent per activity	Activities
All users aggregate time spent per session	Users
All users aggregate time spent per week, month, year	Activities

Table 3. Required API routes for each query for hybrid database structure

Path	Total Downloaded (MB)	Average Downloaded (MB)
/activities	87.2	87.2
/users	86.5	86.5

Figure 7. Hybrid Database Bandwidth Usage Results

Figure 7 above outlines the amount of data downloaded in each of the two paths needed for the hybrid database. Unlike the other two databases, there is only two paths that need to be assessed, and both of them are almost the exact same size. The size of the data is slightly more than the nested database but a lot smaller than the flat database.

Path	Avg Execution Speed (ms)
/activities	110
/users	40

Figure 8. Hybrid Database Speed Results

Lastly, figure 8 shows the average execution speed needed to retrieve the data in the hybrid database. Similar to the bandwidth usage, figure 8 also just uses two paths, with the activities taking up 110ms while the users table only takes up 40ms. They are both fairly low numbers compared to the flat and nested database.

Since the flat database was superior to the nested database in the preliminary testing, the hybrid database is compared against the flat database in the second round of tests. Looking at the bandwidth usage, the hybrid database uses far less bandwidth than the flat database for every type of query other than the one query that requires aggregated sessions, as demonstrated in table 3. For other queries, it uses 86-174 MB. Next, looking at the execution speed, the hybrid databases consistently performs faster than both the nested and flat database structures as it only takes 110ms to go through the activities and states, and minuscule 40ms to go through the users and sessions.

Query	API Call
User's # of sessions per week, month, year	Hybrid
User's time spent on app per week, month, year	Hybrid
User's score progression over time per activity	Hybrid
User's time spent progression over time per activity	Hybrid
All users aggregate activity scores per activity	Hybrid
All users aggregate time spent per activity	Hybrid
All users aggregate time spent per session	Flat
All users aggregate time spent per week, month, year	Hybrid

Table 3. Flat and hybrid database comparison for each query

The hybrid database was measured against the flat database on the 8 common queries, and table 3 above indicates that the hybrid structure performs better on 7/8 of the common queries. This is because the hybrid database does an excellent job of finding the right balance between the nested and flat database by coupling the users and sessions together and the activities and states together. Like the flat database, the separation in the layout makes it a lot more efficient but the size of the data is far less than the flat database, which bolsters its efficiency. Furthermore, the nearly half and half split of the data makes the execution speed a lot more efficient as well.

The main reasoning for the large increase in size for the flat database is all the ids it creates. The separation of the data definitely makes it more efficient than the nested database. However, it also suffers from having to create so many ids to use as foreign keys to connect all 4 tables, which immensely increases the size of the overall database. In contrast, the hybrid database strikes a great balance of splitting the data appropriately so that the database is more efficient but the overall size of the data isn't increased too much either. Specifically, the flat database requires an id on every single of the thousands of states that just consist of two strings otherwise, which is a huge amount of extra data created while the hybrid database doesn't require it. Accordingly, there are too many ids making the database too big in the

flat structure, while the hybrid database just uses one session id as a foreign key to connect the two users and activities tables. Accordingly, this ideal balance between data separation and id creation is demonstrated in table 3, as the hybrid database performs better for all but one query.

VIII. Conclusion

Overall, after thoroughly comparing the flat, nested and newly created hybrid database structures using speed and bandwidth as the two main metrics, it was determined that the hybrid structure was the fastest while using the lowest bandwidth for the majority of the queries. This is due to the fact that it took advantage of the strengths of each of the structures on each opposite end of the spectrum and found the perfect equilibrium in the middle by having two tables connected by one id.

Due to the aforementioned results, the hybrid database structure will be utilized in the Pragmatica product. This will be directly implemented to create the most efficient database to query through the core VR application and the data dashboard web application. This project was extremely useful, as it will massively help optimize the Firebase database for the queries coming in from the data dashboard. This is especially true as the product scales to having a large number of users with years' worth of data that will all be queried, where a massive amount of money can be saved in server costs through the increase in efficiency.

There are a lot of different extensions that can be derived from this project. First of all, the sample data that was used in the experiments was very simple and consistent. All the users had almost the same amount of data with very regular intervals for every timestamp. Therefore, different types of more irregular data could perform very differently with different database structures. The data can be skewed either way or contain certain outliers like some users having a lot more or lot less data. Furthermore, other types of data paradigms instead of session-based data can be tested as well. It would be interesting to test out how the structures perform differently with all this different data. Lastly, other database applications can also be tested as they all have their own intricacies to how they operate, so there may be some unforeseen difference based on the system used. All of these extensions mentioned above are definitely worth exploring in the future to expand on the findings found in this project.