QueueCheck

Real-time Wait Time Monitor for College Campus Services using Cloud-powered NoSQL Database

by Jie Lyu & Junyan Mao

Introduction

Project Title

Real-time Wait Time Monitor for College Campus Services using Cloud-powered NoSQL Database

Team # 3 Members

Jie Lyu - jie.lyu@gatech.edu (Team Lead)

Junyan Mao - jmao@gatech.edu

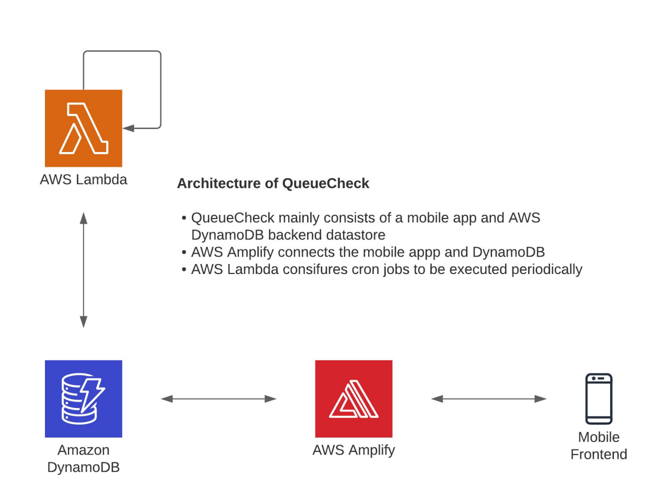
Overall Goal

We want to mitigate the queueing problems on college campuses to

1. Save everybody’s time for waiting in queues
2. Reduce the extent of social gathering to prevent the spread of diseases

Our goal is to make an app to provide users insights for both the current wait time and the statistical wait time chart for different time periods through the day

Architecture



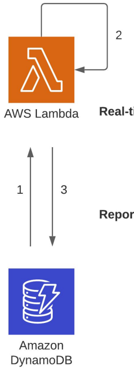
Data Flow

Step 1 - Data points from users are stored on AWS DynamoDB

User Submits a Wait Time Estimation

1. user submits a wait time estimation from mobile app
2. Amplify function trigged to submit a write request to DynamoDB

Step 2 - Cron jobs are periodically triggered to extract insights



Real-time Wait Time Monitoring

1. a cron job is triggered once every 5 minutes to read all estimations from the last 10 minutes in the Estimation table
2. an algorithm aggregates all the estimations
3. the wait time is updated in the Location table

Report Periodically Generated

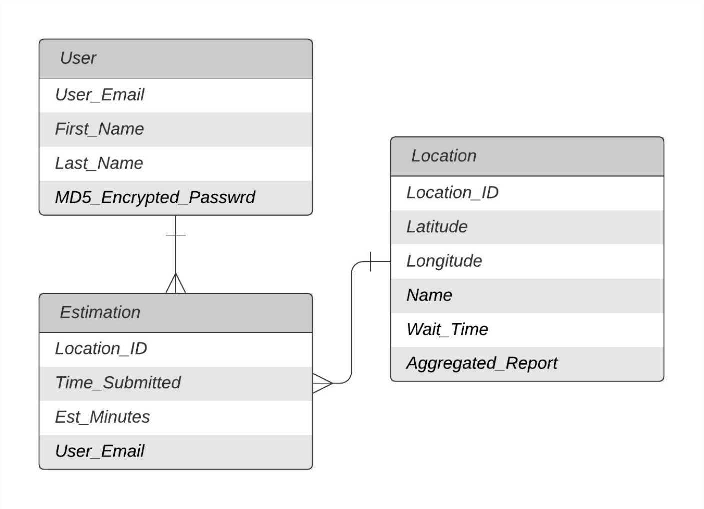
1. a cron job is triggered once every hour to read all estimations from the previous hour in the Estimation table
2. an algorithm aggregates all the estimations
3. the aggregated report is updated in the Location table

Step 3 - Users fetch computed insights directly

User Fetches Insights

1. user opens a location details page from mobile app
2. Amplify function trigged to fetch the real time wait time status and hourly report from DynamoDB
3. insights fetched directly from Location and Report table
4. report displayed on mobile app

Schema Design - Overview



Schema Design - User Table

Definition

User table has **User_Email as its primary key**. Each user will have the password encrypted and saved, and as well as first and last names if provided in registration. This table serve the registration and login functionality.

Performance

Some users will not be willing to tell us their first and last name. A schemaless database like DynamoDB provides such flexibility.

Schema Design - Location Table

Definition

Location table stores all the service locations on campus that we want to monitor. Each location is given an **Location_ID as its primary key**, and contains the name, latitude and longitude for that location. It also contains Wait_Time as the real time wait time status and Aggregated_Report as the hourly wait time report, both updated by AWS Lambda cron jobs.

Performance

Because there are less than 100 locations on campus and their information is unlikely to need to be updated often, and the real time wait time status and aggregated report is only written to this table once every 5 minutes and every hour, the write pressure on the table is not much. On the other hand, it will be read whenever a user views the wait time of a location, so we **only raise the AWS read capacity units, not the write.**

Schema Design - Estimation Table

Definition

Estimation table stores the wait time estimations users submit for a given location. Each estimation record contains the user email, the location ID, the estimated wait time and the time the estimation was submitted.

Performance

The **partition key is set on Location_ID**. Such a partition key does roughly ensure uniform activity across all logical partition keys. Thus, we do not need to create another ID that lead to a waste of space.

A **sort key is set on Time_Submitted**. Even though this table will not be frequently read, an suboptimal read query can still cause stress. Thus, we added a sort key to the Time_Submitted attribute because read queries to this table will likely contain conditionals on that attribute. By adding a sort key, we avoid scanning the whole table for each read query. The composite primary key of Location_ID and Time_Submitted guarantees the uniqueness because Time_Submitted has the granularity of 1 microsecond.

Data

In the real world, all the location data e.g. name, latitude, longitude will be set by administrators, user data e.g. email, name will be provided by users at registration, and wait time estimations will be submitted by users during daily use. For this demo, we will use **pre-defined synthetic data** because it

1. Save time from the cumbersome manual data input
2. Ensure that the data input is consistent between each test
3. Able to adjust the frequency to test out scalability

Technologies

AWS DynamoDB: the main NoSQL database to store everything needed for the app to run

AWS Lambda: defines the two cron jobs to compute and update the real time wait time and aggregated hourly report

Amazon EventBridge: send signals periodically to AWS Lambda to trigger the two cron jobs

AWS Amplify: connects the mobile app backend with AWS DynamoDB

Difficulties

- Very limited documentation and discussion on using AWS Amplify with iOS
- Because we have no previous experience in implementing cron jobs, we spent some time in designing the periodic update pipeline, and had to do some research before we chose AWS Lambda where we implemented the two cron jobs

What We Learnt

- AWS is really powerful. However, while each components seem easy to use, a working app requires communications between different components. It can be hard to figure it out for the first time
- NoSQL is more flexible if we want to expand the current database to add more functionalities and diversities

Future Work

- The algorithm to calculate the real time wait time can be improved by replacing the average function by a weighted time-series function e.g. exponential moving average function (EMA)
- Expand the database to accommodate for more types of data
- Improve the UI of both apps
- Integrate the iOS app and the web app to share the same backend, preferably using AWS