CSCI 5253 Final Project Written Report

Project Name

Shopping history summarization application

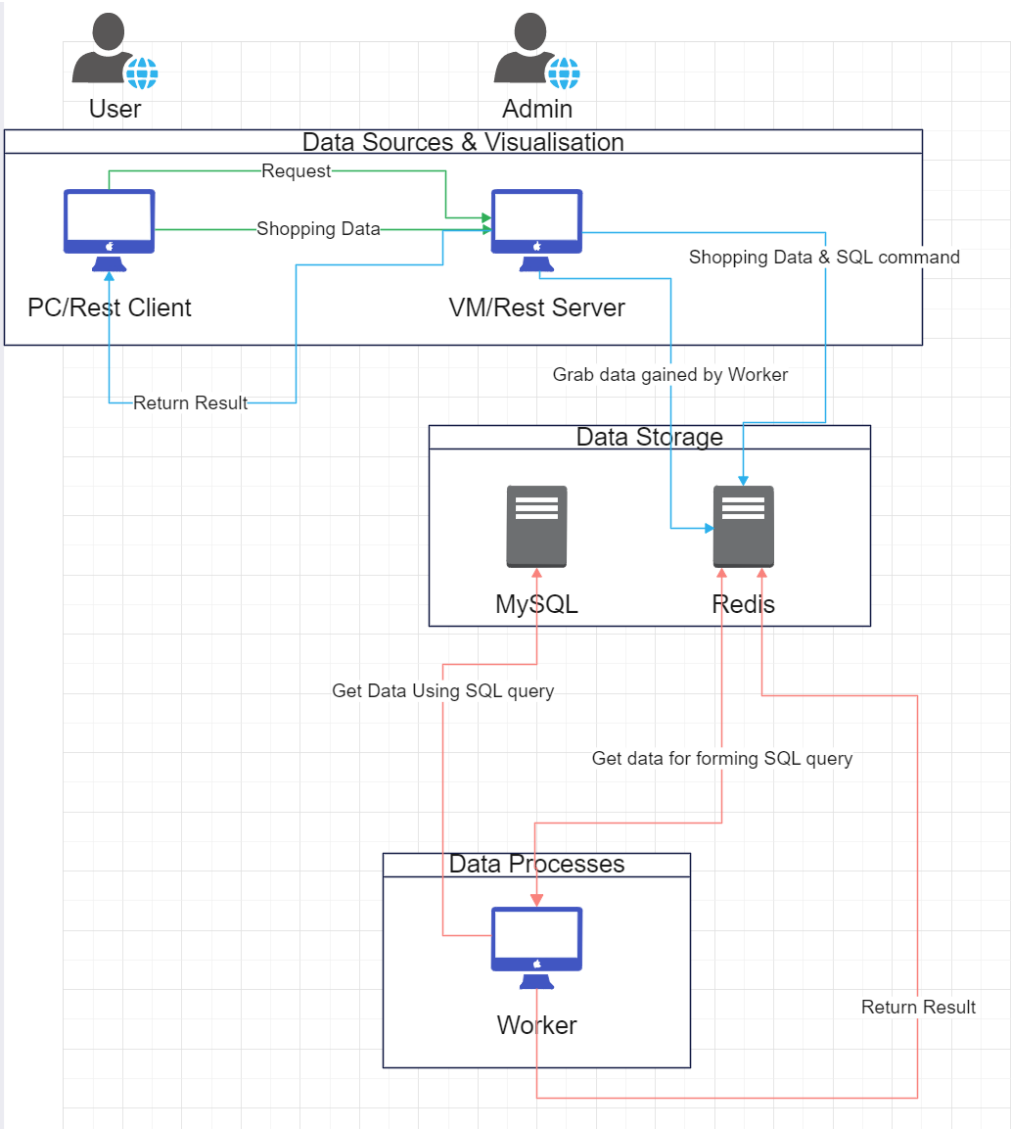
Team Members

Sitong Lu (so “our team”, “we” mentioned in the following description are all referencing to Sitong only)

Project Overview

The goal of our project is to provide a server that can record and summarize the shopping history for the user. This program will allow the user to submit JSON file that records his/her raw shopping history and display the whole history in the way the user ordered. The server will contain other useful methods to allow the program to show the user the most useful information he/she needed on the user interface.

Architecture Diagram



Project Components

The project is containing the following 4 software components (sentences highlighted by blue means goal of the component, green means its advantages, yellow means its disadvantages):

- **API Interface**
  Since the main goal of this project is to allow users to somehow interact with a database that contains their shopping history, an API interface is the most doable way taught from lecture that help to make it possible. By deploying the Flask environment that allows REST server (acting as the remote server) and client (acting as user's machine, e.g., phone, laptop, etc.) to setup, we now have a way to simulate the connection between remote server and user's local machine. Setting up rest server and client, making requests and gaining responds are relatively easy, comparing with gRPC which needs extra proto setting. The only disadvantage I can say is that the working efficiency is not as good as gRPC since the latter one totally relies on its own setup (that's the whole point of setting proto file for gRPC). However, since the project is relatively small and all the processes are not extremely complicated, using REST in this case in not that different comparing with gRPC.
  By receiving response from REST client, REST server recognizes the actual command, grabs all the necessary information that can form a SQL query and push it to REDIS with tag "sql_command". REST server then waits until the command value in REDIS with the corresponding tag has been popped out by Worker node. Now the REST server can grab result from REDIS marked with tag "sql_result" and pass it back to REST client as response.
- **Message queue**
  Because there must be multiple connections between remote server and users' machines, we have to set up a way to let the data processing queries in REDIS, including insertion and deletion of shopping records to be processed one after other depending on the time they were generated. Otherwise, data collision caused by the lag of communication between the server and all different kinds of users' machines may happen, resulting in multiple insertions to the same record or executing a set of queries in the wrong order, hence returning the absolute wrong results to the user. As a given feature, REDIS does support data insertion and deletion in specific order, which can just be treated as a way of executing message queue, hence saving lots of extra time for making one of our own. However, we must be very careful with the existing data still in REDIS, otherwise they may be popped out of the program and used for results that are completely irrelevant.
  By always having programs that need to push or pop data from REDIS use "lpush" and "rpop" commands, we can make sure that message queue has been applied properly, hence making the "first in, first out" procedure for data transition work. In this project, we designed REDIS as a "bus station", allowing it to load/unload data and signal both the REST server and Worker nodes to let them know when they can move on to the next step.
- **Database**
  Storing users' shopping data as a goal set at the beginning of this project means that we definitely need to use a database at the first place. To help us sort and return result back in the way users expected more efficiently instead of simply acting as a data storage, MySQL database was the option the team chose to use in this project. By using SQL queries to grab/edit the database, our team also finished the goal of using the knowledge gained from this course at the

early stages of this semester. However, connecting MySQL database with multiple Python programs means that we need to deal with the SQL database's transaction isolation, a strategy used for protecting the database from been accessed and modified by multiple programs on the same record. The principle of this strategy is good but even though our program already fixed the command collision issue through message queue, SQL database still does not return the newest dataset back to REST server back in time when we were still allowing both the REST server and the Worker node to connect to the same database at the same time, hence making the final response returned to REST client not the most recent one. For fixing this issue, REST server now has stopped connecting to the SQL database and assigned all of its original SQL-related features to Worker node.

Now, the MySQL database only communicates with the Worker node and returns the results to it, which will then push the results to REDIS so that the REST server can fetch them from it.

- **Virtual Machine**

  By hosting all the parts besides REST client on Kubernetes container, we can actually simulate the situation that the server and the client machine are not staying in the same place, proving that the wireless data transmission through the whole program is actually working. To make this possible, our team used VM hosted on Google Cloud. The benefit of using VM on Google Cloud is that all the components besides REST client are in an induvial cluster, making the data transmission and processing more efficient since there's no other program in the container that will slow processes related to the project down (which has been proved even by running Kubernetes cluster on my local PC. It is significantly faster than simply run everything through terminal at the same place). The only disadvantage is that hosting VM on Google Cloud takes money and may charge the team a lot of if we forget to turn the machine down after finishing testing since there are always lots of data been used for hosting these running programs in the cloud.