Map-Reduce With VMs and Cloud Functions

Part-1: Map Reduce on VMs:

The entire flow of running the Map Reduce framework on VMs and computing the Inverted Index has been automated, the entire flow can be triggered with the help of start_map_reduce_on_gcp.sh script.

Implementation Details and Program Flow:

1. Update the ecc.config file specifying the number of mappers, reducers, region, credentials etc. required for the framework.
2. First a VPC is created and required firewall rules are applied.
3. **KV-Store** with Google Cloud Storage as backend is deployed and used as the storage backend for Map Reduce (MR) framework.
4. All the required files are transferred to the kv-store vm and necessary packages are installed using shell scripts. The KV-Store server is deployed and will be accessed through the RPC interface by all the components in the MR Framework.
5. A new VM is spawned for deploying Master Node of the framework. Package installation and required scripts are transferred using bootstrap shell scripts.
6. Now the control is handed over to the Master Node VM.
7. Master Node spawns the required number of Mapper, Reducer VM's and performs bootstrapping and package installations on all the servers.
8. Once all the servers are ready, Master Node deploys Mapper and Reducer processes on respective VMs.
9. Core logic of the framework is similar to the previous assignment. There are no major changes to the application logic apart from the way it is deployed.
10. All the intermediate and final outputs are stored in the KV-Store.
11. Once the Inverted-Index files are generated, the files are transferred to the system from where the entire flow is triggered.
12. Next all the instances are terminated and the clean up process takes place.
    Note: VPC-Subnets, firewall rules are created only once and reused for Part-2.

Part-2: Map Reduce on Cloud Functions:

Design Details:

In this project, the Map Reduce functionality has been achieved using four main functions and Google Firestore as storage, implementation details are as described below:

Functions:

1. Master:
    a. Trigger: HTTP-Trigger
    b. Returns: Success/Failure Message
    c. This function is responsible for handling all the data logistics in the framework.
    d. It should be invoked through a http-post request with the following json payload containing the number of mappers, reducers and the input collection in Google Firestore which has all the input data.

    e. Master fetches data from the input collection, partitions the data and assigns the data to mappers in a round-robin pattern by inserting the partition into another collection which is named after mappers, ex: mapper-0, mapper-1 etc.
    f. Mappers which are also deployed as cloud functions, are triggered by ***cloud-events.*** Mappers perform the mapping task and inform the Master by updating flags in a collection named as “node-status” in firestore.
    g. Master keeps track of all the mappers with the help of node-status flags and assigns tasks to the mappers which are available.
    h. Master waits until all the data is processed by all the mappers and collects the intermediate output written by mappers in another collection named “mapper-output”.
    i. Master then assigns the intermediate mapped data to Reducers by inserting the document id of the intermediate data into another collection named after reducers, ex: reducer-0, reducer-1 etc.
    j. Reducers then writes the final output, the inverted index into a collection named “inverted-index”.
    k. Similar to Mappers, reducers also update their status in “node-status” collection, which enables the Master to keep track of Reducer Nodes.
    l. Once the final output Inverted-Index is generated, the Master responds with a success message as output.

2. Mapper:
    a. Trigger: firestore document.create cloud event.

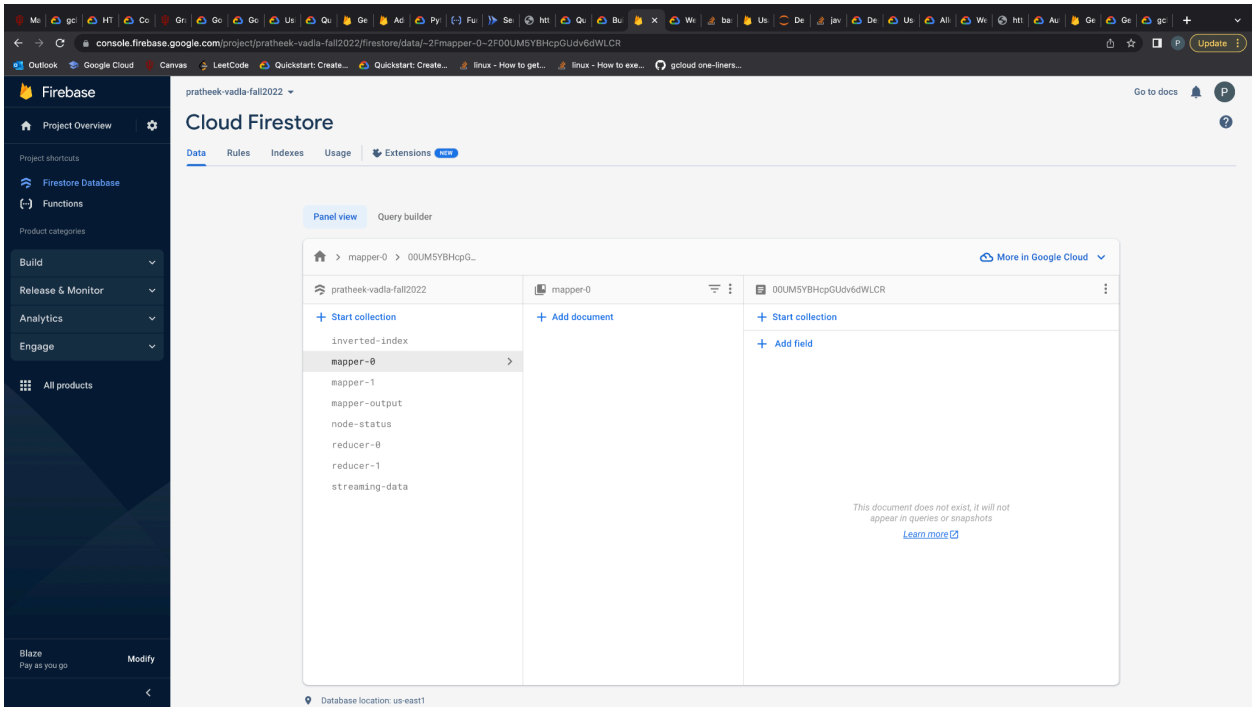
b. Returns: Mapped output in a firestore collection named: “mapper-output”.
c. All the mapper functions keep monitoring for any newly created documents in the collections named after the mappers.
d. If the master wants to assign any text to mapper-0, it inserts that text into a collection named mapper-0. This emits a new cloud-event, triggering the mapper process.
e. Once any mapper is triggered, it first updates its status as busy in the node-status collection. In this way it tells the master that it is busy, so that master assigns tasks accordingly.
f. Mapper then parses the event and fetches the data from the firestore collection, maps the data and inserts into another collection named “mapper-output”. It stores output in the form of a list of dictionaries. Each dict has three keys namely, word, position offset in text and 1.
g. The functionality of ***Distributed Group By*** is handled in the mapper functions. Each mapper, before writing the intermediate output into the firestore collection, generates document id based upon the md5 hash of the first character of the word. In this way, words starting with the same character are all inserted in the same document of the collection.
h. After all the words are processed and the output is written to mapper-output collection, the Mapper functions update the status in node-status collection that it is available for the next job.
i. It is mentioned in the documentation of Firestore event-based triggers that there is no guarantee that the event based triggers occur only once. There is a chance that the same trigger might be delivered twice, which leads to processing of the same text multiple times by multiple mapper nodes.
j. To handle the above issue, master adds a flag in each entry called ***“is_processed”.*** Once any mapper has picked up this trigger and processes this text, it updates the is_processed flag to true. This will ensure that the same document is not processed by other mappers.

3. <u>Reducer:</u>
    a. Trigger: firestore document.create cloud event.
    b. Returns: Reduced Inverted Index in a firestore collection named: “Inverted-Index”.
    c. Master collects document ids from intermediate mapper-output collection and then assigns these doc ids to Reducers by inserting them into collections named after reducers.
    d. Similar to the mappers, each reducer monitors their respective collections/documents and starts reducing the data once any event is emitted.
    e. To ensure idempotency a “is_processed” flag based mechanism is implemented in the reducers as well.
    f. Reducers after finishing their jobs, update their statuses in node-status collection to inform the master that their job is completed.

4. Streaming-Data-Indexer:
    a. Trigger: firestore document.create cloud event.
    b. This function is responsible for indexing newly added documents.
    c. This function keeps monitoring for events emitted by documents present in the collection named “streaming-data”.
    d. When any new event is emitted, the streaming-data function parses the event, context and triggers the indexing process.
    e. All the above explained steps will be executed sequentially.

Storage:

1. This project is designed with the help of Google Firestore.

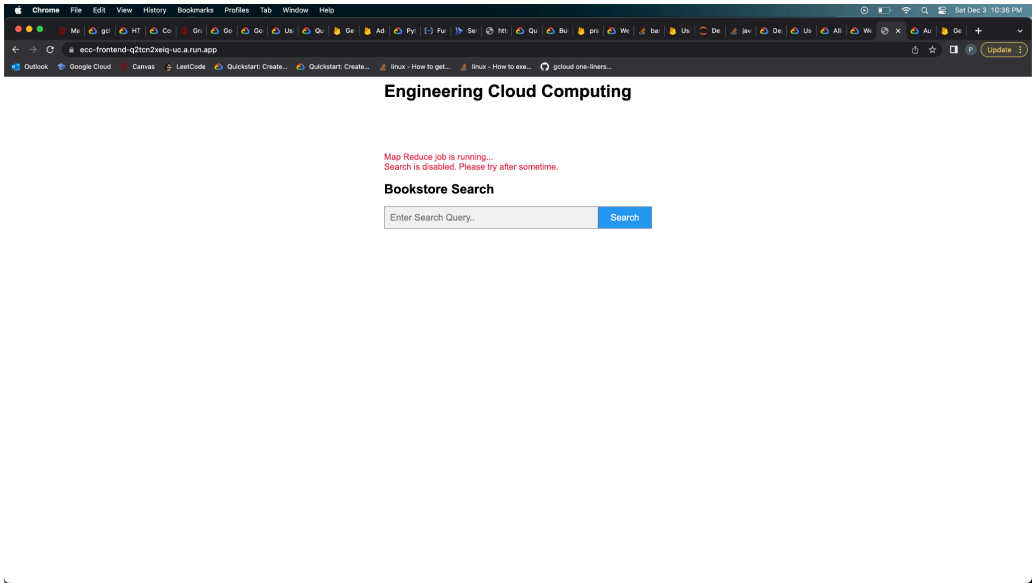


2. As shown in the above picture, the firestore has several collections, designed to handle different components of the system.
3. Input Data Collection, this has the input data which will be uploaded by the user. There is a script named “upload_to_firestore.py” which can be used to upload the files from local storage to firestore. There are no restrictions on the naming convention of the input data collection. User needs to remember the name of this collection as this will be used by the master function initially to create the inverted index.
4. Mapper Collections: Each mapper has its own collection named after itself. The mappers keep monitoring these collections for any newly created events. If any new document is created, immediately the respective mapper starts processing the data.
5. Reducer Collections: Similar to mapper collections, these are also named after reducers which monitor for new documents.
6. Mapper Output: This is the collection for storing intermediate output of Mapper Nodes.

7. Node Status: The node-status collection is one of the most important collections in the firestore. As all the nodes update their status/readiness/availability in this collection. Each mapper, reducer has documents after their names, which will be used to track the specific entities state.
8. Node Status also has flags like job_status, which tells if there is any indexing process which is currently running. This helps the frontend component to freeze the search functionality until the indexing process is finished.
9. Streaming Data: This collection is responsible for updating the index when any new document is added.
10. Inverted Index: This is the collection where the final index is stored.

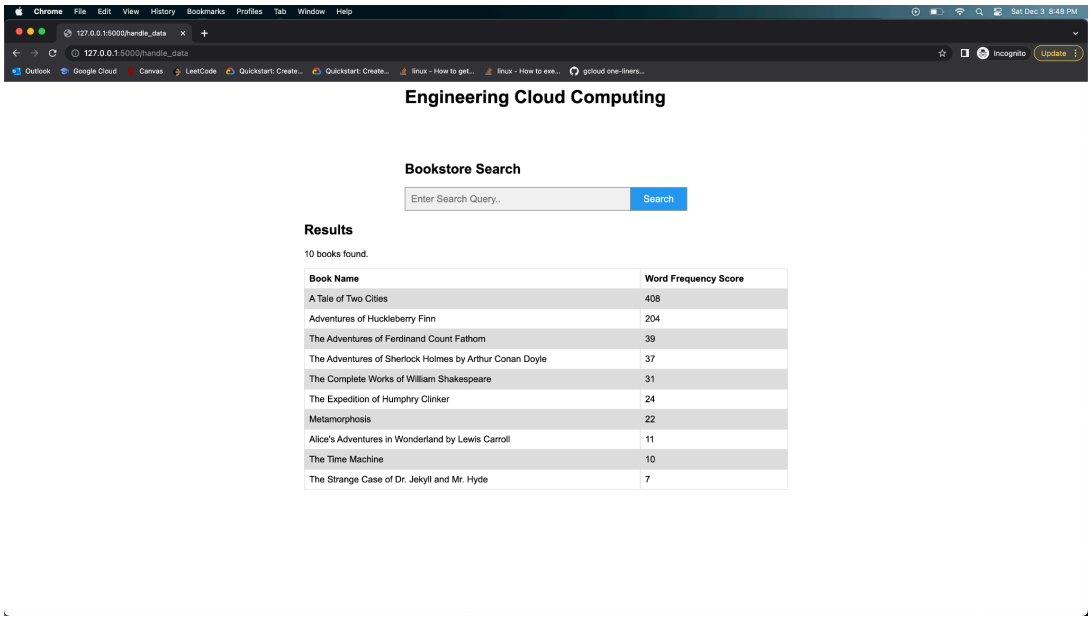
Frontend:

1. The search interface is developed using flask.



During indexing the above message is displayed, and the search functionality is disabled until the completion of the indexing process.

2. Search Results are displayed in tabular format as shown below:

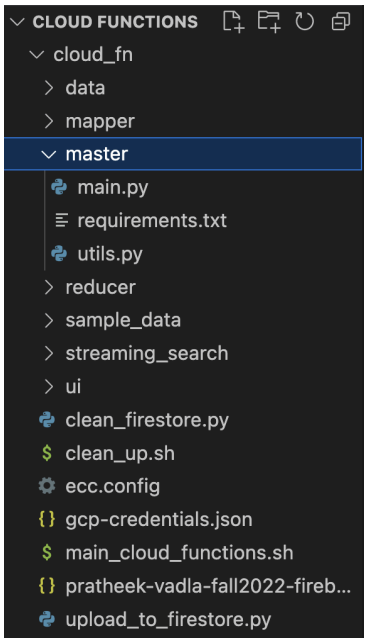


The results are sorted in the decreasing order of their frequency scores.

3. The frontend/Search application is deployed through Google Cloud Run.
4. Backend functionality is mainly a set of queries to the Inverted-Index collection in the firestore.

Steps to Execute the code:

1. The scripts for each component have been stored in separate folders.



2. The entire flow of the framework has been automated and can be triggered with the help of main_cloud_functions.sh bash script.
3. Edit the ecc.config file to update the configuration details.
4. Run ./main_cloud_functions.sh to trigger the framework.
5. Run ./clean_up.sh to clean all the deployments.
   Note: Please run this script after Part-1 as VPC Networks, Firewalls are only created once in Part-1 and re-used for Part-2.
6. Please use upload_to_firestore.py script for uploading files to Cloud Firestore. This script takes input as firebase credentials json, input folder path and uploads all the files present in the given path.
   Usage: python3 upload_to_firestore.py --input_path=data --collection=ecc-data-v2 --credentials=pratheek-vadla-fall2022-firebase-adminsdk.json

Cost Estimate:

1. In this implementation, I have used single instances of all types of functions each having 256MB of memory and .167 vCPU.
2. Typical costing for this configuration is around: $0.000000463 per 100ms
3. I have observed that for 15 books, the entire end-to-end flow takes around 2.5-3 mins.
4. Assuming that it takes 3 mins, the total cost incurred to compute one inverted index is 3 x 3600 x 10 x 0.000000463 = $0.05.