Homework Coding Test Solution

Written by: Camilo Fandino

Email: Sergio.camilo.fandino@gmailcom

1. Introduction

This document presents a comprehensive exposition of the development process for a tailored solution to a specified problem. It is designed to walk readers through the end-to-end journey of crafting this solution, from conceptualisation to the final design. It is important to note that while this document outlines a structured approach and potential solution, not all components may be fully operational or production-ready. Some sections serve an illustrative purpose, demonstrating theoretical implementations or methodologies that may require further adaptation and rigorous testing before deployment in a production environment. The core objective of this narrative is to provide clear insights into the problem-solving strategies and engineering considerations that were employed in devising this solution.

2. Description of the Problem

DataMart creation, there will be new account data of customer shared from Open Banking. Bank may access customer account data in other banks once we got consent from Customer. Once consent removed, data would be removed from source table. There is a need to have a DataMart which may accelerate analysis of customer asset in other banks.

Table: **TBL_OB_ACCOUNT**
Refresh Frequency: Real-time
PK: BANK_CODE, REAL_ACNO

FIELDS	DESCRIPTION	TYPE
BANK_CODE	Account banking code	STRING
REAL_ACNO	Real account number from other bank. Primary key.	STRING
REAL_CUST_ID	Customer real customer id	STRING
BALANCE	Account Balance in account currency	NUMERIC
CURRENCY	Account currency	STRING
UPDATE_DTTM	Data update timestamp	timestamp

Table: **TBL_FX_RATE**
Refresh Frequency: Daily historical

FIELDS	DESCRIPTION	
FROM_CURRENCY	Foreign Exchange from currency code	STRING
TO_CURRENCY	Foreign Exchange to currency code	STRING
EXCHANGE_RATE	Exchange rate	DOUBLE
ACCURACY	Number of accurate digits after decimal point while exchange	INTEGER
TIMESTAMP	Data update timestamp	timestamp

Process Thinking

Within this section, I will focus on explaining what would be the way I normally would tackle a problem given the concrete tasks and a defined problem. The requirements will be just briefly mentioned, since they were stated already in the received document.

- *Problem Understanding:* Understanding the problem and define it properly is important, consulting stakeholders and checking twice what is the intention and business needs is crucial to design a solution.
    - We need to implement a DataMart that enables rapidly data consumption of customer account data from multiple banking institutions.
    - I would like to clarify with the stakeholder what is exactly "that" what we want to achieve, I would like to firstly find the business question that we are trying to answer with the implementation; Having more context help me not just to provide a fitted solution, but also helps me by designing my solution in a way that we are able to be one step ahead of the data consumer in case they will bring more requirements in

the near future. e.g. Which other banks may be strong competitors based on the amount of assets they hold from the current bank customers?

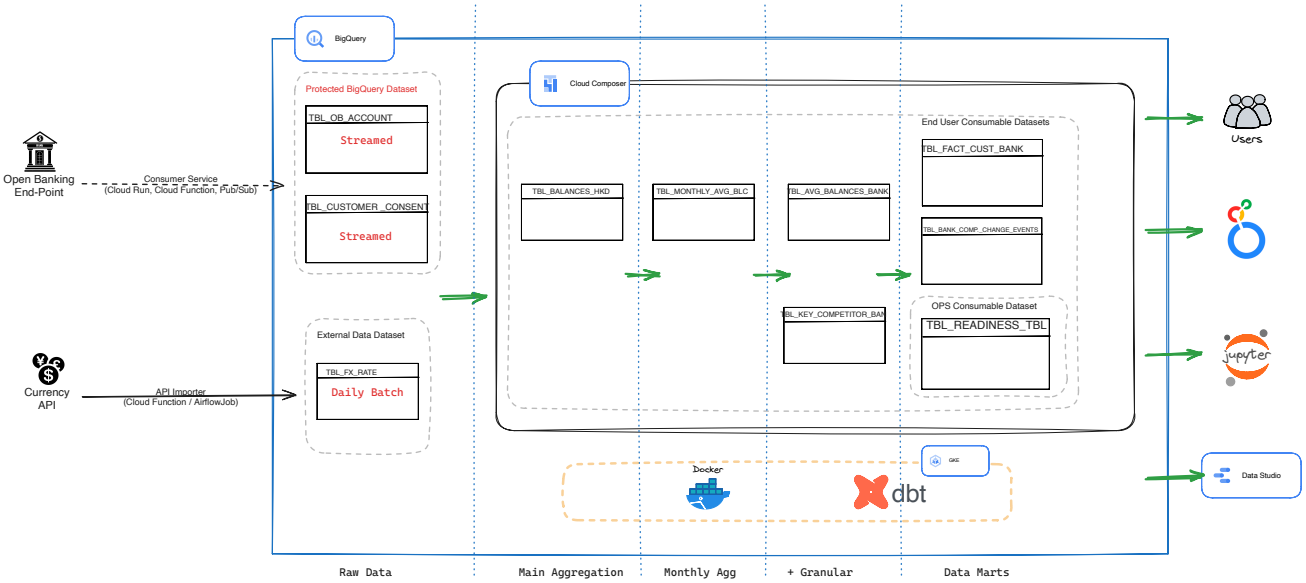
- I also would like to clarify the structure and the data source from the consumer consent data - this should be probably answered by the stakeholders.
- It is always crucial to sit together and collaborate with colleagues to get different point of views and experience, I normally do this, saves a lot of time.
- Note: Here I will normally clarify other doubts related to the business as well related to the ingested available data - important to define deadlines and align on expectations, e.g. When can we schedule the next meeting to align on progress? When can we deliver a test DataMart for perhaps data analysts/data consumers? (Important that other teams can be unblocked in case they need to build dashboards, draft their models etc..)

- *Requirements Definition:*
  - Functional Requirements
    - Data Anonymisation/Pseudonymization
    - Consent tracking
    - Normalised granular data at user level for analytical purposes
    - Right to be forgotten implementation
  - Non-Functional Requirements
    - ETL must consider real-time data capabilities
    - Data Quality measurements
    - Alerting & Monitoring
    - Technical Documentation (Data schemas flow/ER)
    - Data User Documentation (Data Catalog)
    - Deployable Pipeline (IAAC)
    - Tests implementation (Unit tests, framework, etc…)
- *Assumptions*
  - I assume that there is at least one component already ingesting data into the tables named by the task and this is not part of the current homework.
  - I assume data is being ingested into the same data warehouse where the end DataMart will be located
  - The data for user consent it is as well being imported through the Open Banking connection
- *Next Steps*
  - Solution Design
    - Data Flow
    - ER diagram
    - SQL Scripts
  - Technical Decisions
  - Improvements Further Steps

3. Solution Design

Data Flow

The following diagram is an approximate of the data flow and architecture of the solution. Below a few details about the components and how could a implementation look like.



- Data Ingestion (Data Sources):

  Open Banking data is consumed/pulled through a secured connection already facilitated and integrated through a reliable service. This service acts as a mediator, ensuring that data is ingested in a controlled and scalable manner - time is ingested in real time.

  Concurrently, currency exchange rates are imported via an API Importer, typically a Cloud Function or an Airflow job, to fetch the latest rates that are crucial for currency normalization across different bank accounts, the job would run once daily.

- Initial Storage in BigQuery:

  Raw data from both sources lands in a protected BigQuery Dataset, ensuring that sensitive information is handled with the utmost security therefore segregating the keys solicited by the task. This dataset contains the TBL_OB_ACCOUNT table, and the TBL_OB_CONSENT table that are just accessed either by an authorised service account or a group of users that have the role & permissions for it. Note that no end user will have access to this dataset.

  The TBL_FX_RATE table resides in another Dataset, serving as a repository for the exchange rate information fetched from the Currency API.

- Pipeline Architecture:

  I am going for a daily batching approach, where regardless of the real time data streaming the pipelines are transforming and loading the tables once a day. Nonetheless, there is the chance of providing a Table View that could do the computations near-real time, or even reduce the jobs to a micro-batching type. But according to the requirements, it seems that the insights are not business critical. Also worth mentioning that having a near real time pipeline incur in high computing costs, so it is worth analyse the ROI value for the data.

- Transformation of the data:

The raw data undergoes its first transformation to ensure privacy compliance through pseudonymisation, I use in this case the SHA256 algorithm provided by BigQuery to hash the columns as requested by the requirements . This process generates a version of the data where direct identifiers are replaced with hashed values and are consistent for analysis purposes.

I created different aggregations that allows later on to present the requested measures and that covers the requirements, as for example the AVG of balance of the customer in HKD - I achieve this by joining information from the consent table and the currency table, to also be able to bring the dates of the first and end dates of consent given by the user.

I am also creating a historical table where I have stored by user an event every time a user has a new TOP_AVG_BANK value to fulfil one of the requirements of the task, I do this with a window function that allows me to store the old top bank and the new one, this in one single query.

- Cloud Composer and dbt:

  I am leveraging Airflow to orchestrate the sequence of jobs, managing dependencies and ensuring that data is processed in the correct order, as well for triggering 2 important tasks:

  - Running a job after each SQL script is run that inserts a row inside the TBL_DATA_READINESS table
  - Running a daily job that checks the TBL_OB_CUSTOMER_CONSENT table and based on the SQL logic, it deletes the rows that match the users that do not provide consent (and alternative would be to replace the ids with another hashing algorithm not used previously)

- Data Consumption:

  Finally, the data flows to the end users through various interfaces and tools. Users can access the transformed and ready-to-use data via BigQuery for complex queries, Jupyter notebooks for exploratory analysis and machine learning tasks, and Google Data Studio/Looker for visualisation and reporting.

ER Diagram

TBL_OB_CUSTOMER_CONSENT

Description: Maintains records of customer consents for accessing their banking data.

Attributes:

REAL_CUST_ID: Customer real customer id.

BANK_CODE: Account banking code.

CONSENT: A Boolean flag indicating whether consent has been given.

EVENT_TIME: The timestamp of when the consent event occurred.

TBL_BALANCES_HKD

Description: This table holds all the balances in HKDf from the TBL_OB_ACCOUNT

Attributes:

BANK_CODE: Account banking code.

HASHED_REAL_CUST_ID: A hashed version of the customer id.

HASHED_REAL_ACNO: A hashed version of the account number.

BALANCE_HKD: The account balance converted to Hong Kong Dollars.

UPDATE_DTTM: The timestamp of the creation of the balances.

TBL_MONTHLY_AVG_BALANCES_BANK_CUSTOMERS

Description: This table holds the monthly average of balances of customers per bank

Attributes:

HASHED_REAL_CUST_ID: A hashed version of the customer id.

BANK_CODE: Account banking code.

YEAR_MONTH: The Year and Month of the balance.

AVG_BALANCE_HKD: Monthly average of balances excluding Sundays

TBL_AVG_BALANCES_BANK_CUSTOMER

Description: This table holds the average balances calculated per customer per bank.

Attributes:

BANK_CODE: Account banking code.

HASHED_REAL_CUST_ID: A hashed version of the customer id.

INGESTION_TIMESTAMP: Timestamp of the ingested record

AVG_BALANCE_HKD_OVERALL: Average of monthly balances of a customer

TBL_KEY_COMPETITOR_BANK_CUSTOMER

Description: This table hold ranked average balance per customer per bank

Attributes:

HASHED_REAL_CUST_ID: A hashed version of the customer id.

TOP_AVG_BANK: Account banking code

INGESTION_TIMESTAMP: Timestamp of the ingested record

TOP_BANK_BALANCE: The average amount of the balance.

RANKING: Ranking for identifying average held balance per bank from a customer

TBL_BANK_COMPETITOR_CHANGE_EVENTS

Description: Tracks historical changes in the customer’s preferred bank based on the highest average balance.

Attributes:

HASHED_REAL_CUST_ID: A hashed version of the customer id.

OLD_BANK_CODE: Name of the previous top bank competitor

NEW_BANK_CODE: Name of the new Bank top competitor

CHANGE_TIMESTAMP: Timestamp of the occurred change

TBL_FACT_CUSTOMER_BANK

Description: A fact table that consolidates customer account information for analytics.

Attributes:

HASHED_REAL_CUST_ID: A hashed version of the customer id.

BANK_CODE: Account banking code.

AVG_BALANCE_HKD: The average account balance converted to Hong Kong Dollars.

FIRST_CONSENT_DATE: The date when the customer first consented to data sharing.

EXIT_CONSENT_DATE: The date when the customer withdrew consent, if applicable.

TOP_AVG_BANK: The bank code of the institution where the customer has the highest average balance.

UPDATE_DTTM: The timestamp of the last update to this record.

TBL_DATA_READINESS

Description: Indicates the readiness of the data for use by data consumers.

Attributes:

HASHED_REAL_CUST_ID: A hashed version of the customer identifier for privacy.

BANK_CODE_TOP_COMPETITOR: The bank code of the competitor bank.

EVENT_DATE: The date when the event of changing the top competitor bank occurred.

Note: The TBL_OB_ACCOUNT and TBL_FX_RATE are not described here, since they are already discarded and part of the given task.

4. Scripts

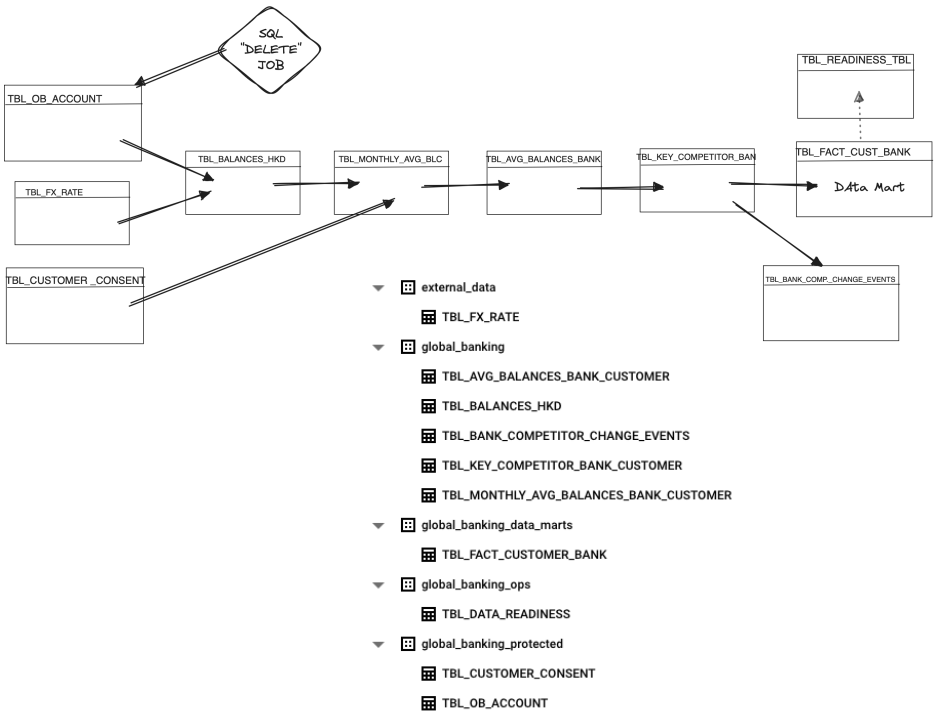
I have written and attached different SQL scripts, in a production environment they are part of the deployment. If we consider for this solution, we could have them stored in a Version Control repository like GitHub and we could then deploy them into a Bucket Storage and run the SQL scripts from the Airflow Composer.

Another way we could use the SQL scripts, would be to use them inside the dbt tool, that at the same time puts the scripts in a framework-organised way.

In both ways, the scripts would then need to be Jinja templated, so we could add dynamism to the ingestion of the data.

Note: The scripts are in different files, and also include the jobs that run for example when the consent of a customer is not given anymore (SQL "DELETE" job).

The files represent the following structure:



5. Technical Decisions

Cloud Provider

- We will work with Google Cloud Platform -> Is the cloud provider from HSBC

Data Storage - Analytics

- BigQuery

  Performance: Optimized for fast querying on large datasets

  Scalability: It handle large datasets efficiently and cost-effectively, scaling automatically with your data volume.

  Serverless architecture: Serverless architecture, eliminating server management and simplifying scaling.

  Cost-effectiveness: Pay-per-use pricing model of BigQuery, making it cost-efficient for varying data workloads.

  Integration with Open Banking: Showcase BigQuery's pre-built connectors for various APIs and data sources, facilitating easy integration with Open Banking

Job Orchestration

- Composer (Airflow managed):

  Managed Airflow: It is a secure and well supported tool, it is additionally managed focusing on reduced operational overhead.

  DAG scheduling and orchestration: Works well as scheduler and able to manage complex data pipelines with its DAG structure.

  Monitoring and logging: Built-in monitoring and logging capabilities - easy pipeline debugging and maintenance.

Data Transformation

- dbt:

  Declarative approach: Declarative SQL language that simplifies data transformation logic, improving code readability and maintainability.

  Modularization and reusability: It has modularity of dbt models, allows code reusability and promotes best practices.

  Version control and collaboration: It has integration with version control systems like Git for easy versioning and collaboration.

Version Control & CI/CD

- GitHub:

  Industry standard: Emphasize the widespread adoption of GitHub for version control, ensuring familiarity and ease of use for developers.

  Collaboration features: Highlight the collaboration features of GitHub like code reviews and issue tracking, facilitating teamwork and code quality.

  Integration with CI/CD tools: Mention the seamless integration of GitHub with various CI/CD tools, creating a smooth development workflow.

- Cloud Build

  Native GCP integration: Streamlined data flow, pre-built steps, shared authentication.

  Scalability & Cost: Serverless architecture, pay-per-use, built-in caching.

  Ease of Use & Collaboration: Simple configuration, Git integration, GitHub compatibility.

  Security & Reliability: GCP security features, continuous integration, monitoring & logging.

- Terraform

  Consistent configuration: Ensures consistent infrastructure configuration across environments (dev, test, prod) reducing errors and simplifying deployments

  CI/CD integration: Integration with Cloud Build for automated infrastructure provisioning and configuration changes

  Security: Enforces security and best practices through code reviews and automated deployments to reduce risks.

Improvements and Further Steps

There are other factors that are worth mentioning and I could have work if this were a real project, or also future steps:

- If it is true, that when writing the Data Warehouse model, I was thinking not to duplicate the raw table TBL_OB_ACCOUNT and extend the table with the hashed IDs in order to make the joins and reduce the complexity, I went for an approach that even tho it covers all the requirements, it made the work harder for myself.
- Important to always check existent name conventions - adjust GitHub project names, dataset names, table names, field names, etc…)
- Jenkins or GitHub Actions could be also two really good tools, but it depends of the team knowledge to use them or not.
- There are Data Quality checks that need to be put in place; dbt could help here, but also a good tool could be Great Expectations, that helps to validate data and also notify data users about issues.
- It is on the table to analyse if a dbt managed version (dbt Cloud) could benefit the organisation, rather than maintaining a GKE cluster for dbt, in the long term this will require work hours from an IC.
- The work together with other data engineers, infrastructure engineers and SREs is crucial to bring deployments in a stable manner and build a reliable platform where advanced data users can create and expand data pipelines easily.w
- A way of alerting data users for data readiness, would be implementing a task in Airflow that either send an alert to a communication system like Slack, Teams etc…
- The SQL scripts must be templated to be able to use the partitioning future from BigQuery, this would reduce costs when reading big tables and also improve the performance.
- It is necessary to have in place a strict Roles and Permissions structure applied not just to BigQuery (datasets and tables) but also to deployments, where the deployment of keys should be also well engineered.
- I could look for clustering fields in a few tables depending on performance. For example BANK_CODE wise.