Data Quality Checks

Assuming the current data size to be the one that was sent to me, I would recommend Pandas for data quality checks.

pandas: Suitable for medium-sized datasets

Spark: Ideal for terabyte-scale data processing.

General Assumptions

- All datasets are loaded as pandas DataFrames.
- Each dataset contains specific columns pertinent to its domain.
- Valid currency codes are defined as `['EUR', 'USD', 'GBP', 'NOK', 'SEK']`.
- The exchange rate should be a positive float value.
- The `ingest_date_time` column is added to each dataset to track the date and time of data ingestion.

Data Validation Checks

1. Accounts Data Validation

The following validation checks are performed on the `accounts_df` dataframe:

1.1 Missing Values Check

- Description: Ensures there are no missing values in the dataframe.
- Action: Collects rows with missing values and records an error comment: "Missing values in the row".

1.2 Duplicate Account Numbers on the Same Reference Date Check

- Description: Ensures there are no duplicate account numbers on the same reference date.

- Action: Collects duplicate rows and records an error comment: "Duplicate account number on the same reference date".

1.3 Negative Amount Check

- Description: Ensures that the 'amount' column does not contain negative values.
- Action: Collects rows with negative amounts and records an error comment: "Negative amount".

1.4 Valid Date Check

- Description: Ensures that the 'reference_date' column contains valid dates.
- Action: Collects rows with invalid dates in the 'reference_date' column and records an error comment: "Invalid date in 'reference_date'".

1.5 Valid Account Number Check

- Description: Ensures that the 'account_number' column contains valid integers.
- Action: Collects rows where 'account_number' contains non-integer values and records an error comment: "'account_number' contains non-integer values".

1.6 Valid Amount Check

- Description: Ensures that the 'amount' column contains valid floats.
- Action: Collects rows where 'amount' contains non-float values and records an error comment: "'amount' contains non-float values".

1.7 Valid Account Type Check

- Description: Ensures that the 'account_type' column only contains 'Asset' or 'Liability'.
- Action: Collects rows where 'account_type' contains invalid values and records an error comment: "'account_type' contains invalid values".

1.8 Valid Account Name Check

- Description: Ensures that the 'account_name' column follows the expected pattern.

- Action: Collects rows where 'account_name' contains invalid values and records an error comment: "'account_name' contains invalid values".

2. Deposits Data Validation

The following validation checks are performed on the `deposits_df` dataframe:

2.1 Missing Values Check

- Description: Ensures there are no missing values in the dataframe.
- Action: Collects rows with missing values and records an error comment: "Missing values in the row".

2.2 Duplicate Customer Check

- Description: Ensures there are no duplicate customers based on the 'customer' column.
- Action: Collects duplicate rows and records an error comment: "Duplicate customer".

2.3 Negative or Zero Amount Check

- Description: Ensures that the 'amount' column does not contain negative or zero values.
- Action: Collects rows with negative or zero amounts and records an error comment: "Amount is negative or zero".

2.4 Valid Date Check

- Description: Ensures that the 'start_date', 'maturity_date', and 'reference_date' columns contain valid dates.
- Action: Collects rows with invalid dates in any of these columns and records an error comment: "Invalid date in 'column_name'".

2.5 Valid Currency Check

- Description: Ensures that the 'currency' column contains valid currency codes.
- Action: Collects rows with invalid currency codes and records an error comment: "Invalid currency".

2.6 Valid Exchange Rate Check

- Description: Ensures that the 'exchange_rate' column contains valid floats and values greater than 0.
- Action: Collects rows with invalid or non-positive exchange rates and records an error comment: "Invalid exchange rate".

3. Loans Data Validation

The following validation checks are performed on the `loans_df` dataframe:

3.1 Missing Values Check

- Description: Ensures there are no missing values in the dataframe.
- Action: Collects rows with missing values and records an error comment: "Missing values in the row".

3.2 Duplicate Customer Check

- Description: Ensures there are no duplicate customers based on the 'customer' column.
- Action: Collects duplicate rows and records an error comment: "Duplicate customer".

3.3 Negative or Zero Amount Check

- Description: Ensures that the 'amount' column does not contain negative or zero values.
- Action: Collects rows with negative or zero amounts and records an error comment: "Amount is negative or zero".

3.4 Valid Date Check

- Description: Ensures that the 'start_date', 'maturity_date', and 'reference_date' columns contain valid dates.
- Action: Collects rows with invalid dates in any of these columns and records an error comment: "Invalid date in 'column_name'".

3.5 Valid Currency Check

- Description: Ensures that the 'currency' column contains valid currency codes.
- Action: Collects rows with invalid currency codes and records an error comment: "Invalid currency".

3.6 Valid Exchange Rate Check

- Description: Ensures that the 'exchange_rate' column contains valid floats and values greater than 0.
- Action: Collects rows with invalid or non-positive exchange rates and records an error comment: "Invalid exchange rate".

An observation that I made is that the customer and reference date together is likely the Primary key but to be on the safer side I have moved the duplicated customer records to the Invalid data folder. Ideally will work with Business to understand what validations need to be made.

Data Modelling

Databases are a good fit for Online Transactional Processing / OLTP systems. Whereas Data warehouses are the best fit for Online Analytical Processing / OLAP systems

Normalization which is the process of diving a big table into small tables using 1NF, 2NF, and BCNF is a good idea for reducing redundancy if the primary goal is making the writing operation efficient but in our case, where we are modeling for a data warehouse, our primary goal is to make querying efficient

Can reporting be carried out with OLTP systems?

Yes, but not the optimal choice as OLTP systems are not meant for analyzing large volumes of data. They are meant for handling transactional data. Here are the reasons why an OLTP system cannot be a good fit for reporting/ analysis purposes

- Since Reporting would require a lot of data to get insights, it would
involve a lot of joins operations if it were to be an OLTP system (consisting

of Normalized small tables). Joins are costly operations.

- It could overload the OLTP system as it has to perform complex joins
and analyze large volumes of data, thereby making it less performant
even with the transactional tasks for which it is meant.

Data warehouses DWH : (Online Analytical Processing - OLAP) are the best fit for such reporting or analysis purposes. OLAP systems are most performant and optimized for Read queries.

Reporting Database Design

- The intent of this design is to support the business queries and not
meant for data maintenance.

-The resulting model reflects the kind of questions that the business
wants to ask and not on the functions of the underlying operational
System.

- **Performance Focus: on retrieving the data quickly (faster reads)**
- Larger datasets are retrieved as a result of a query.
- Insert and update speeds are not relevant.

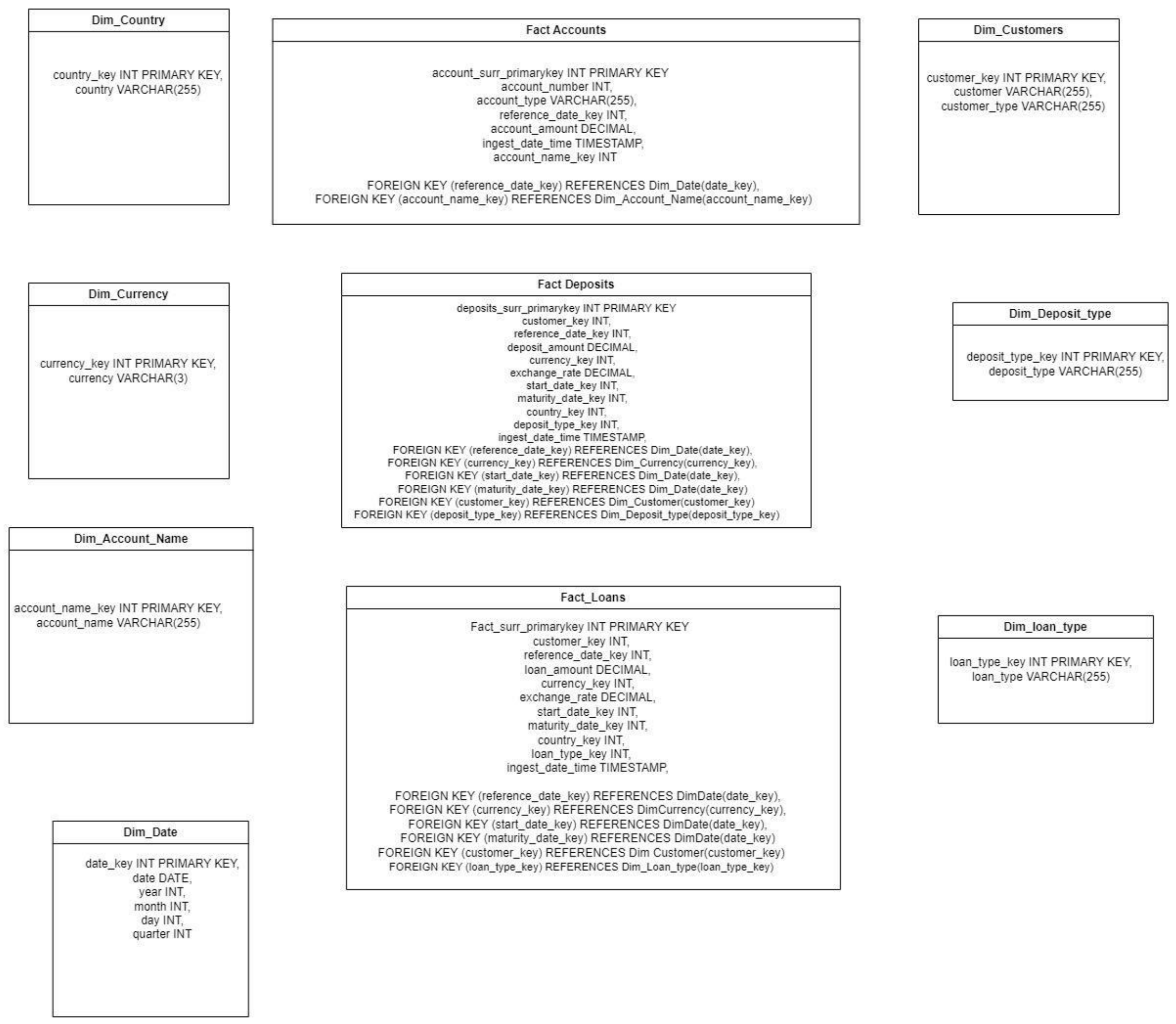
Dimensional Modelling (STAR schema) is a better approach chosen for the following reason

1. Facts and Dimensions are different tables
2. In case we want to add loan types or any other dimensions over time it will be easier to do so as it is just a matter of creating a dimension table and adding the Primary key to the fact table (Updates for increasing dimensions will not be very painful as dimension tables are smaller than fact tables as the data grows)
3. Not only Optimization is important in our case what is also important is Business should be able to understand the models without having detailed knowledge of underlying performance considerations and Facts and Dimensions make it simpler for Businesses to Understand
4. Controlled De Normalization by using a STAR-like schema is useful for faster querying Faster Reads at the cost of slower Updates/Inserts

Dividing our table columns into facts and dimensions based on the required **granularity.** The choice of this granularity (**Grain** as it is called in Kimball's terms) will decide how the dimension tables will be split

Here is the Data model STAR Schema

https://app.diagrams.net/#G1fIyoVaFBs4iqo9w7-mJd26672CLsRrOx#%7B%22pageId%22%3A%22l8cUQMvowJF55cJT_aG4%22%7D



```
Dim_Country
country_key INT PRIMARY KEY,
country VARCHAR(255)
```

```
Fact Accounts
account_surr_primarykey INT PRIMARY KEY
account_number INT,
account_type VARCHAR(255),
reference_date_key INT,
account_amount DECIMAL,
ingest_date_time TIMESTAMP,
account_name_key INT

FOREIGN KEY (reference_date_key) REFERENCES Dim_Date(date_key),
FOREIGN KEY (account_name_key) REFERENCES Dim_Account_Name(account_name_key)
```

```
Dim_Customers
customer_key INT PRIMARY KEY,
customer VARCHAR(255),
customer_type VARCHAR(255)
```

```
Dim_Currency
currency_key INT PRIMARY KEY,
currency VARCHAR(3)
```

```
Fact Deposits
deposits_surr_primarykey INT PRIMARY KEY
customer_key INT,
reference_date_key INT,
deposit_amount DECIMAL,
currency_key INT,
exchange_rate DECIMAL,
start_date_key INT,
maturity_date_key INT,
country_key INT,
deposit_type_key INT,
ingest_date_time TIMESTAMP,
FOREIGN KEY (reference_date_key) REFERENCES Dim_Date(date_key),
FOREIGN KEY (currency_key) REFERENCES Dim_Currency(currency_key),
FOREIGN KEY (start_date_key) REFERENCES Dim_Date(date_key),
FOREIGN KEY (maturity_date_key) REFERENCES Dim_Date(date_key)
FOREIGN KEY (customer_key) REFERENCES Dim_Customer(customer_key)
FOREIGN KEY (deposit_type_key) REFERENCES Dim_Deposit_type(deposit_type_key)
```

```
Dim_Deposit_type
deposit_type_key INT PRIMARY KEY,
deposit_type VARCHAR(255)
```

```
Dim_Account_Name
account_name_key INT PRIMARY KEY,
account_name VARCHAR(255)
```

```
Fact_Loans
Fact_surr_primarykey INT PRIMARY KEY
customer_key INT,
reference_date_key INT,
loan_amount DECIMAL,
currency_key INT,
exchange_rate DECIMAL,
start_date_key INT,
maturity_date_key INT,
country_key INT,
loan_type_key INT,
ingest_date_time TIMESTAMP,

FOREIGN KEY (reference_date_key) REFERENCES DimDate(date_key),
FOREIGN KEY (currency_key) REFERENCES DimCurrency(currency_key),
FOREIGN KEY (start_date_key) REFERENCES DimDate(date_key),
FOREIGN KEY (maturity_date_key) REFERENCES DimDate(date_key)
FOREIGN KEY (customer_key) REFERENCES Dim Customer(customer_key)
FOREIGN KEY (loan_type_key) REFERENCES Dim_Loan_type(loan_type_key)
```

```
Dim_loan_type
loan_type_key INT PRIMARY KEY,
loan_type VARCHAR(255)
```

```
Dim_Date
date_key INT PRIMARY KEY,
date DATE,
year INT,
month INT,
day INT,
quarter INT
```

I have deliberately not created arrows but instead written the foreign key to primary key relations between the facts and the dimensions.

Design Decisions

If the exchange rate was relatively stable I would put it in the currency dimension table but here I decided to keep it in the fact table Also I did not create a dimension table for account_type as there are only 2 values ('Assest', 'Liability') and they are unlikely to change moreover, for account_name I have created a dimension table as there is possibility for more analytics on certain account_names. (Unsecured Personal Loan - EUR, Credit Card - NOK..etc)

I have created a Surrogate PRIMARY Key for every Dimension table and Fact table which is required to support the implementation history for slowly changing dimensions. Having a surrogate primary key for Fact tables is also helpful in identifying each unique record

To choose the Dimensional Model, it is very important to know the **granularity** of the reporting, and user stories can be helpful for example:

Neil makes a deposit of 1000 dollars on the date 31st Dec 2023 in the currency EUR from Finland

Questions: who? how much? what date? …. etc Therefore STAR schema was chosen for flexibility

Datawarehouse choice:

The popular data warehousing solutions include

1. AWS redshift
2. Google Big Query
3. Azure Snapsis Analytics
4. Snowflake
5. Apache Hive compatible with HDFS (On-prem)

Comparison summary (Generated by ChatGPT)

Feature	**Apache Hive**	**Amazon Redshift**	**Google BigQuery**	**Azure Synapse Analytics**	**Snowflake**
Scalability	High	High	High	High	High
Performance	Moderate (requires tuning)	High	High	High	High

Management	Requires setup and maintenance	Fully managed	Fully managed	Fully managed	Fully managed
Cost	Generally cost-effective	Can be expensive	Pay-per-query can be expensive	Can be expensive	Complex pricing can be high
Integration	Hadoop ecosystem	AWS ecosystem	Google Cloud ecosystem	Azure ecosystem	Multi-cloud
Ease of Use	Complex	Moderate	Easy	Moderate	Easy
Flexibility	High	Moderate	Moderate	High	High

Comparison of On-Premises Data Warehousing Solutions (Generated by Chat GPT)

Feature	**Apache Hive**	**Apache Impala**	**Apache HAWQ**	**ClickHouse**	**Greenplum**	**Apache Druid**
Performance	Moderate	High	High	Very High	High	Very High
Scalability	High	High	High	High	High	High
Ease of Use	Moderate	High	Moderate	Moderate	Moderate	Moderate
Integration	Hadoop Ecosystem	Hadoop Ecosystem	Hadoop Ecosystem	Standalone, but versatile	Hadoop Ecosystem	Standalone, but versatile

Query Language	HiveQL (SQL-like)	SQL (ANSI SQL)	SQL (PostgreSQL-based)	SQL	SQL (PostgreSQL-based)	SQL (Druid Query Language)
Storage Format	Supports multiple (Text, ORC, Parquet, etc.)	Supports multiple (Parquet, ORC, etc.)	Supports multiple (HDFS, etc.)	Columnar storage	Row and columnar storage	Columnar storage
Community Support	Large and active	Active, smaller than Hive	Smaller, less active	Growing, active community	Moderate, active community	Growing, active community
Real-Time Processing	No	No	No	Yes	No	Yes
Resource Management	YARN	YARN	YARN	Requires custom setup	YARN	Requires custom setup
Ease of Setup	Complex	Moderate	Complex	Moderate	Complex	Moderate
Cost	Open-source, hardware costs	Open-source, hardware costs	Open-source, hardware costs	Open-source, hardware costs	Open-source, hardware costs	Open-source, hardware costs
Maintenance	High	Moderate	High	Moderate	High	Moderate

In-Memory Processing	No	Yes	No	Yes	No	Yes

Apache Hive

- Performance: Moderate, suitable for batch processing and ETL tasks. Requires tuning for complex queries.
- Scalability: Can handle very large datasets across distributed Hadoop clusters.
- Ease of Use: Moderate; HiveQL is similar to SQL but requires an understanding of Hadoop.
- Integration: Seamless integration with the Hadoop ecosystem.
- Community Support: Large and active community with extensive documentation.
- Real-Time Processing: Not designed for real-time processing, better for batch processing.
- Resource Management: Uses YARN for resource management.
- Ease of Setup: Complex setup due to Hadoop dependencies.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

Apache Impala

- Performance: High; optimized for low-latency, interactive SQL queries.
- Scalability: Can scale to large datasets but can be resource-intensive.
- Ease of Use: High; uses ANSI SQL, making it accessible for users familiar with SQL.
- Integration: Integrates well with Hadoop and other big data tools.
- Community Support: Active community, but smaller than Hive.
- Real-Time Processing: Not designed for real-time streaming data.
- Resource Management: Uses YARN for resource management.
- Ease of Setup: Moderate; easier than Hive but still requires Hadoop.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

Apache HAWQ

- Performance: High; leverages PostgreSQL for advanced SQL capabilities.
- Scalability: Scales well with Hadoop integration.
- Ease of Use: Moderate; based on PostgreSQL, but requires understanding of Hadoop.
- Integration: Integrates with Hadoop and HDFS.
- Community Support: Smaller community, less active compared to Hive and Impala.
- Real-Time Processing: Not designed for real-time processing.
- Resource Management: Uses YARN for resource management.
- Ease of Setup: Complex; requires significant configuration.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

ClickHouse

- Performance: Very High; optimized for real-time analytical queries with columnar storage.
- Scalability: Scales horizontally with ease.
- Ease of Use: Moderate; requires understanding of ClickHouse’s SQL and configuration.
- Integration: Standalone, but can integrate with various data sources and sinks.
- Community Support: Growing and active community.
- Real-Time Processing: Yes; designed for real-time data ingestion and fast querying.
- Resource Management: Requires custom setup for resource management.
- Ease of Setup: Moderate; simpler than Hadoop-based solutions.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

Greenplum

- Performance: High; optimized for OLAP workloads with PostgreSQL base.
- Scalability: Scales well across distributed systems.
- Ease of Use: Moderate; requires knowledge of PostgreSQL and distributed systems.
- Integration: Integrates with Hadoop and other data sources.
- Community Support: Moderate; active but smaller community.
- Real-Time Processing: Not designed for real-time processing.
- Resource Management: Uses YARN for resource management.
- Ease of Setup: Complex; significant configuration required.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

Apache Druid

- Performance: Very High; designed for real-time data ingestion and fast querying.
- Scalability: Scales horizontally with ease.
- Ease of Use: Moderate; requires understanding of Druid's architecture and query language.
- Integration: Standalone, but versatile integration options.
- Community Support: Growing and active community.
- Real-Time Processing: Yes; supports real-time data ingestion and querying.
- Resource Management: Requires custom setup for resource management.
- Ease of Setup: Moderate; setup is simpler than Hadoop-based solutions but still requires configuration.
- Cost: Open-source; costs are primarily for hardware and operational overhead.

Conclusion

The best on-premises data warehousing solution depends on your specific needs:

- For Batch Processing and Integration with Hadoop: Apache Hive is a strong choice, especially if you already have a Hadoop ecosystem.
- For Low-Latency, Interactive Queries: Apache Impala offers high performance for interactive SQL queries.
- For Real-Time Analytics: ClickHouse and Apache Druid are excellent choices due to their real-time processing capabilities and fast query performance.
- For Advanced SQL Capabilities on Hadoop: Apache HAWQ leverages PostgreSQL features for complex SQL queries.
- For High-Performance OLAP Workloads: Greenplum provides strong OLAP capabilities with a PostgreSQL base.

Given so many technologies, our application is for batch processing and data in millions of new rows per year, growing at 10% per year, for the next 5 years keeping scalability, cost-effectiveness, and known query pattern in mind, I would select Google Big Query. I have chosen Google Big Query for scalability and performance and also cost-effective as it is pay as you use model (serverless architecture), also it integrates with different technologies within IT systems

To deal with the quoted problem from the business case "*Inconsistent data quality. There will be times when parts or the entire data history need to be rerun due to faulty data in the source, please consider this*."

1. An incremental Data loading pipeline is built to see if there are updates in any of the dimension tables and incremental surrogate keys are then mapped back to the fact tables to include these updates before putting them into the warehouse

2. As all dimension tables have additional columns start_date, end_date, and active_flag to keep track of all active records in the data warehouse

3. A deleted dimension in the source (a very unlikely scenario):

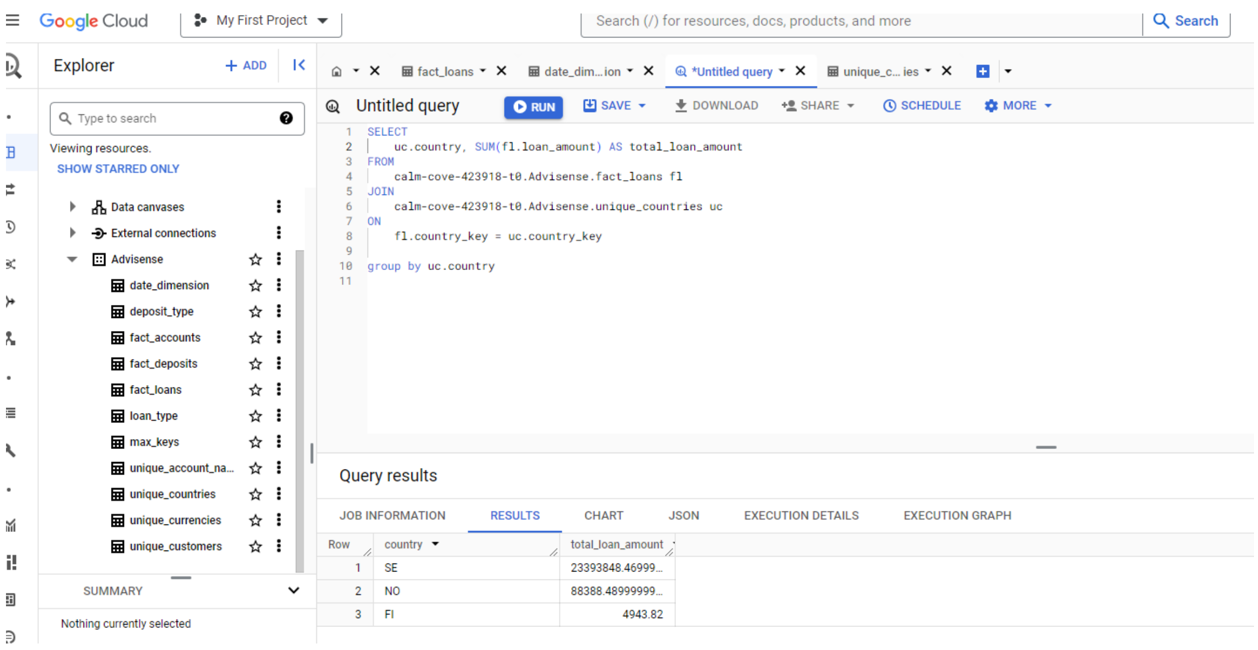
   Example:

   In the first data load taking the CSV files converting them into data models and putting them into Dataware house we will have fresh data equal to the source but if after some time, we have some changes in the source data how can we track history and reflect those changes in data warehouse.

   Let’s say in the source we have a deleted dimension then one way to deal with it is to make the active_flag = ‘N’ for all records with that particular dimension, How to identify those records just take a target to source full outer join and the records on source side which are nulls will be the deleted ones

4. Fact tables have surrogate Primary keys and for every load, the max surrogate keys for the fact tables are persisted so that every new fact record can be uniquely identified without relying on system-generated keys

<u>Implementation</u>



```sql
SELECT
    uc.country, SUM(fl.loan_amount) AS total_loan_amount
FROM
    calm-cove-423918-t0.Advisense.fact_loans fl
JOIN
    calm-cove-423918-t0.Advisense.unique_countries uc
ON
    fl.country_key = uc.country_key

group by uc.country
```

Row	country	total_loan_amount
1	SE	23393848.46999...
2	NO	88388.48999999...
3	FI	4943.82