"Analyzing Customer Financial Behaviors: A Study of Account Balances, Types, and Transaction Activities" USING SQLite.

NAME: DEBORAH SERKI GIWA

STUDENT ID: 22027116

Abstract

This report would delve into the financial patterns and customer behaviors within a hypothetical banking institution. The primary focus would be on understanding the relationship between account types and customer engagement as measured by transaction activities and account balances. Here are the key sections that could be covered.

Introduction

- Provides an overview of the dataset and the simulation approach used to generate the data.
- Discusses the purpose and significance of analyzing customer data in banking.

.

Data Generation Methodology

The dataset was created using Python, with NumPy and Pandas libraries to simulate a realistic set of banking records. Multiple tables were established to facilitate a thorough analysis..

Data Types, Values and python code used to generate the customer table

- **Nominal Data:** The 'Name' and 'Email' columns contain nominal data—categorical data without a natural order. 'Name' combines a randomly chosen first and last name from predefined lists, while 'Email' is constructed using the name with a randomly chosen email domain.

- **Interval Data:** The 'Age' column provides interval data, representing numerical values without a true zero point.
- **Ratio Data:** The 'Balance' column includes monetary values and is considered ratio data since it has a true zero point, and the ratios between values are meaningful.

Random Generation within Sensible Ranges:

- **CustomerID**: Unique five-digit customer IDs ensure each customer has a distinct identifier. This also serve as the Primary Key

- **Names:** The 'Name' field combines a randomly chosen first and last name from available lists to simulate a variety of full names.

- **Emails:** Email addresses are crafted by converting names to lowercase, splitting them, and combining them with a randomly selected email domain, yielding plausible addresses.

- **Genders** are assigned by randomly selecting from the three given options, reflecting a simplified representation of gender diversity.

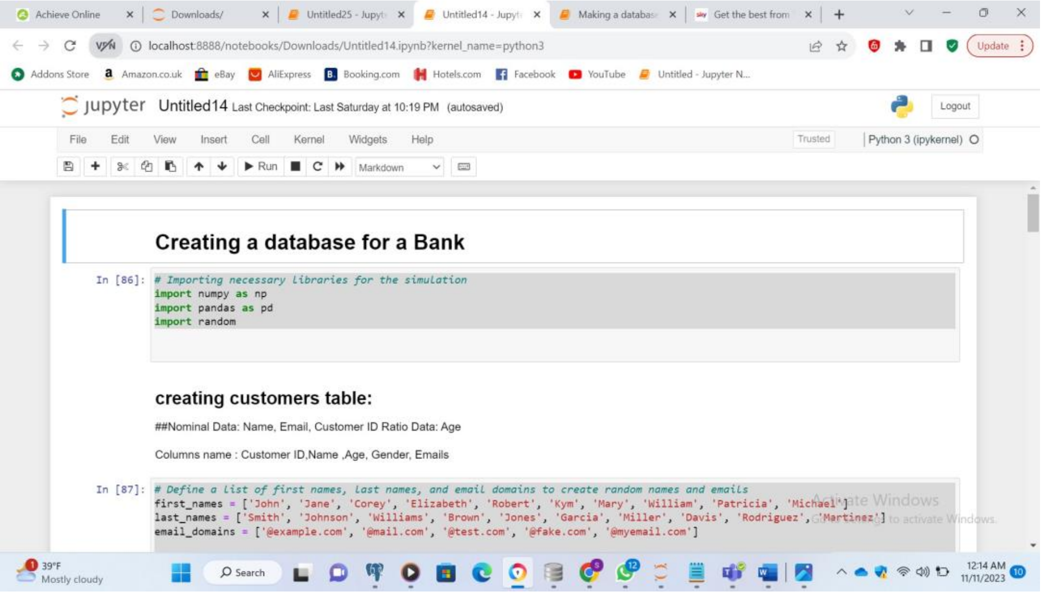
<u>Handling of Missing Values:</u>

- **Emails:** The script introduces missing values (NaN) into the 'Email' column to mimic the common scenario of incomplete data by randomly assigning NaN to 5% of the entries.

<u>Rows and Columns:</u>

- The dataset consists of 1000 rows, each representing a unique customer.
- The columns in the dataset include CustomerID, Name, Age, Gender, and Email

"**Sample Code Blocks:** Customers table generation



Creating a database for a Bank

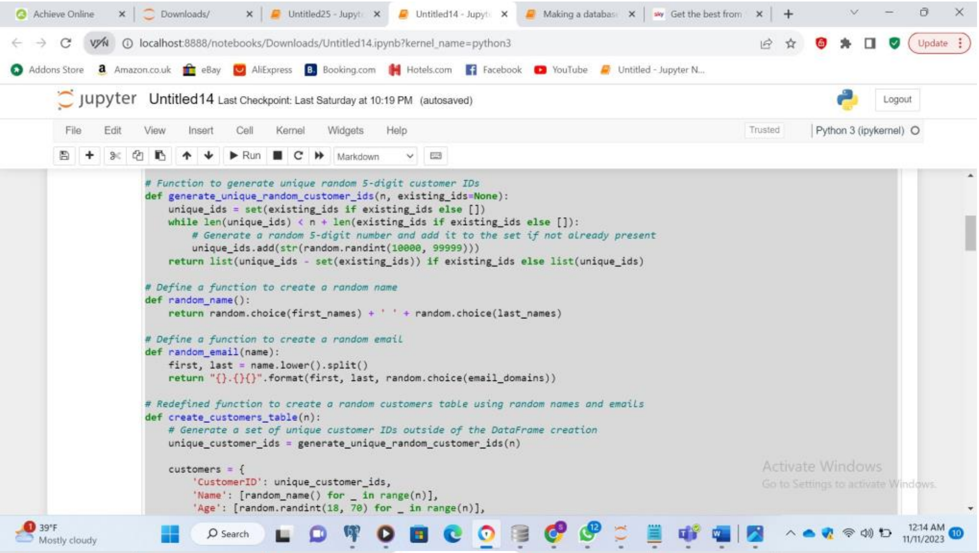
In [86]:
```
# Importing necessary libraries for the simulation
import numpy as np
import pandas as pd
import random
```

creating customers table:

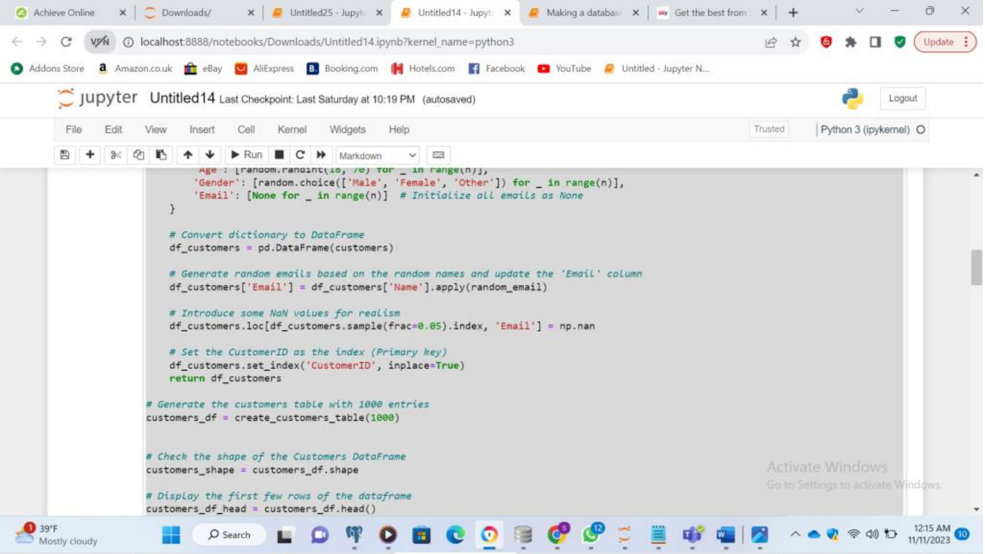
##Nominal Data: Name, Email, Customer ID Ratio Data: Age

Columns name : Customer ID,Name ,Age, Gender, Emails

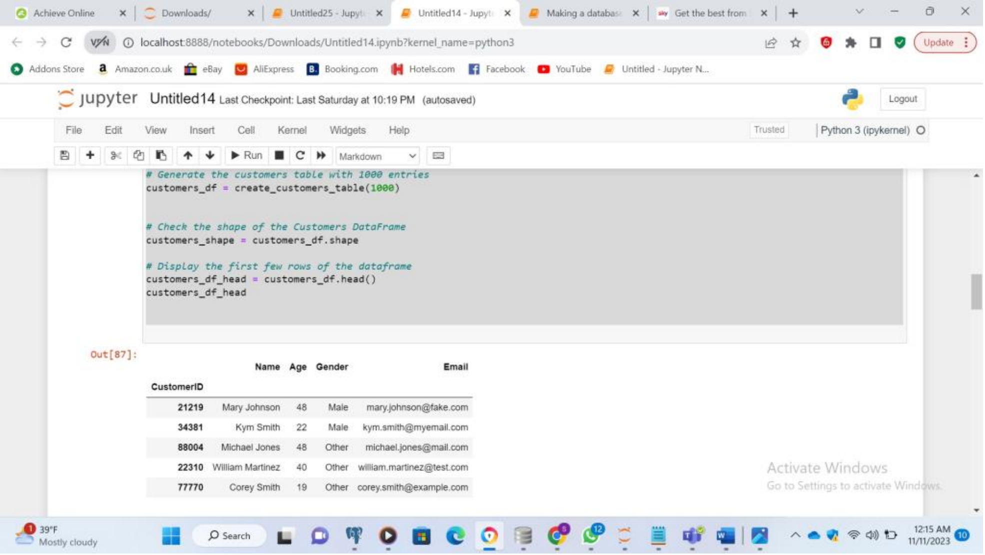
In [87]:
```
# Define a list of first names, last names, and email domains to create random names and emails
first_names = ['John', 'Jane', 'Corey', 'Elizabeth', 'Robert', 'Kym', 'Mary', 'William', 'Patricia', 'Michael']
last_names = ['Smith', 'Johnson', 'Williams', 'Brown', 'Jones', 'Garcia', 'Miller', 'Davis', 'Rodriguez', 'Martinez']
email_domains = ['@example.com', '@mail.com', '@test.com', '@fake.com', '@myemail.com']
```



```python
# Function to generate unique random 5-digit customer IDs
def generate_unique_random_customer_ids(n, existing_ids=None):
    unique_ids = set(existing_ids if existing_ids else [])
    while len(unique_ids) < n + len(existing_ids if existing_ids else []):
        # Generate a random 5-digit number and add it to the set if not already present
        unique_ids.add(str(random.randint(10000, 99999)))
    return list(unique_ids - set(existing_ids)) if existing_ids else list(unique_ids)

# Define a function to create a random name
def random_name():
    return random.choice(first_names) + ' ' + random.choice(last_names)

# Define a function to create a random email
def random_email(name):
    first, last = name.lower().split()
    return "{}.{}{}".format(first, last, random.choice(email_domains))

# Redefined function to create a random customers table using random names and emails
def create_customers_table(n):
    # Generate a set of unique customer IDs outside of the DataFrame creation
    unique_customer_ids = generate_unique_random_customer_ids(n)

    customers = {
        'CustomerID': unique_customer_ids,
        'Name': [random_name() for _ in range(n)],
        'Age': [random.randint(18, 70) for _ in range(n)],
```



```python
        'Age': [random.randint(18, 70) for _ in range(n)],
        'Gender': [random.choice(['Male', 'Female', 'Other']) for _ in range(n)],
        'Email': [None for _ in range(n)]  # Initialize all emails as None
    }

    # Convert dictionary to DataFrame
    df_customers = pd.DataFrame(customers)

    # Generate random emails based on the random names and update the 'Email' column
    df_customers['Email'] = df_customers['Name'].apply(random_email)

    # Introduce some NaN values for realism
    df_customers.loc[df_customers.sample(frac=0.05).index, 'Email'] = np.nan

    # Set the CustomerID as the index (Primary key)
    df_customers.set_index('CustomerID', inplace=True)
    return df_customers

# Generate the customers table with 1000 entries
customers_df = create_customers_table(1000)


# Check the shape of the Customers DataFrame
customers_shape = customers_df.shape

# Display the first few rows of the dataframe
customers_df_head = customers_df.head()
```



```python
# Generate the customers table with 1000 entries
customers_df = create_customers_table(1000)


# Check the shape of the Customers DataFrame
customers_shape = customers_df.shape

# Display the first few rows of the dataframe
customers_df_head = customers_df.head()
customers_df_head
```

Out[87]:

	Name	Age	Gender	Email
CustomerID				
21219	Mary Johnson	48	Male	mary.johnson@fake.com
34381	Kym Smith	22	Male	kym.smith@myemail.com
88004	Michael Jones	48	Other	michael.jones@mail.com
22310	William Martinez	40	Other	william.martinez@test.com
77770	Corey Smith	19	Other	corey.smith@example.com

Data Types, Values and python code used to generate the Account table

Columns and Data Types:

- **AccountNumber**: A unique 8-digit number for each account, representing nominal data, as it categorizes each account without implying any quantitative value.

- **CustomerID**: A reference to the unique customer ID from the 'Customers' table. This serves as a foreign key and is also nominal.

- **AccountType**: A categorical variable representing the type of account. Since the types are ordered (Basic, Standard, Premium), this can be considered ordinal data.

- **Balance**: A numerical value representing the amount of money in the account. This is ratio data since it has a true zero point and allows for meaningful comparisons and calculations.

- **DateOpened**: The date the account was opened, formatted as a string but representing an interval data type because the time between any two dates is meaningful.

Random Generation and Sensible Ranges:

- **AccountNumber**: Randomly generated within the range of 10000000 to 99999999, ensuring that each account has a unique and realistic account number.

- **AccountType**: **AccountType:** Selected from a predefined list to reflect typical bank account categories.

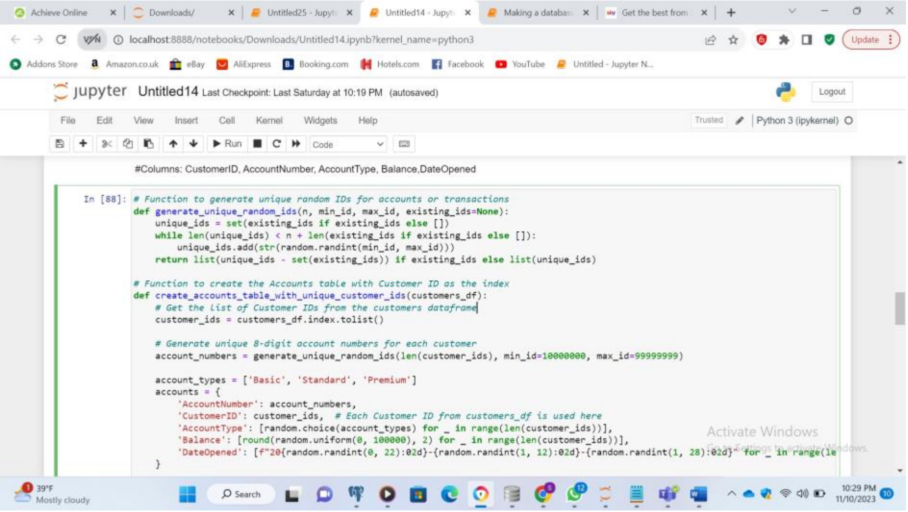
- **Balance**: Randomly generated to be any number between 0 and 100,000, rounded to two decimal places. This range is broad enough to include a wide variety of possible account balances, from empty accounts to those that are quite substantial.

- **DateOpened:** Dates range from 2000 to 2022, representing plausible account opening years.

Handling of Missing Values:

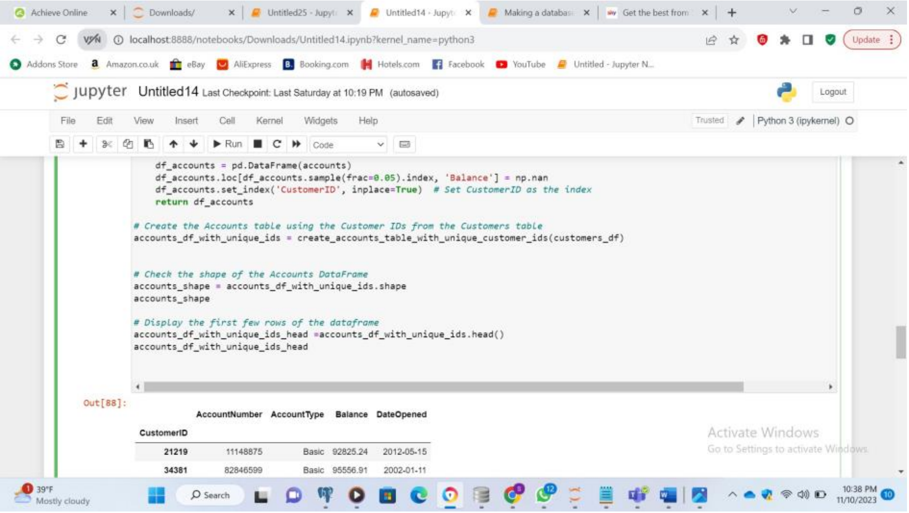
- **Balance**: Introduces NaN for 5% of accounts to simulate instances of unknown or unrecorded balances.

"Sample Code Blocks : Account table generation


jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

#Columns: CustomerID, AccountNumber, AccountType, Balance,DateOpened

```
In [88]: # Function to generate unique random IDs for accounts or transactions
def generate_unique_random_ids(n, min_id, max_id, existing_ids=None):
    unique_ids = set(existing_ids if existing_ids else [])
    while len(unique_ids) < n + len(existing_ids if existing_ids else []):
        unique_ids.add(str(random.randint(min_id, max_id)))
    return list(unique_ids - set(existing_ids)) if existing_ids else list(unique_ids)

# Function to create the Accounts table with Customer ID as the index
def create_accounts_table_with_unique_customer_ids(customers_df):
    # Get the List of Customer IDs from the customers dataframe
    customer_ids = customers_df.index.tolist()

    # Generate unique 8-digit account numbers for each customer
    account_numbers = generate_unique_random_ids(len(customer_ids), min_id=10000000, max_id=99999999)

    account_types = ['Basic', 'Standard', 'Premium']
    accounts = {
        'AccountNumber': account_numbers,
        'CustomerID': customer_ids,  # Each Customer ID from customers_df is used here
        'AccountType': [random.choice(account_types) for _ in range(len(customer_ids))],
        'Balance': [round(random.uniform(0, 100000), 2) for _ in range(len(customer_ids))],
        'DateOpened': [f"20{random.randint(0, 22):02d}-{random.randint(1, 12):02d}-{random.randint(1, 28):02d}" for _ in range(le
    }
```


jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

```
    df_accounts = pd.DataFrame(accounts)
    df_accounts.loc[df_accounts.sample(frac=0.05).index, 'Balance'] = np.nan
    df_accounts.set_index('CustomerID', inplace=True)  # Set CustomerID as the index
    return df_accounts

# Create the Accounts table using the Customer IDs from the Customers table
accounts_df_with_unique_ids = create_accounts_table_with_unique_customer_ids(customers_df)


# Check the shape of the Accounts DataFrame
accounts_shape = accounts_df_with_unique_ids.shape
accounts_shape

# Display the first few rows of the dataframe
accounts_df_with_unique_ids_head =accounts_df_with_unique_ids.head()
accounts_df_with_unique_ids_head
```

Out[88]:

	AccountNumber	AccountType	Balance	DateOpened
CustomerID				
21219	11148875	Basic	92825.24	2012-05-15
34381	82846599	Basic	95556.91	2002-01-11

Data Types, Values and python code used to generate the Transaction table

'Transactions' table that is associated with the 'Accounts' table via the **`CustomerID`**. Here is an analysis of the 'Transactions' table:

Columns and Data Types:

- **TransactionID**: A unique identifier for each transaction, typically a numerical value treated as nominal data.
- **CustomerID**: A foreign key linking each transaction to a specific customer, also nominal data.
- **TransactionType**: A categorical variable indicating the type of transaction, which is nominal data.

- **Amount**: A numerical value representing the money involved in the transaction, which is ratio data because it has a true zero point and allows for meaningful arithmetic operations.
- **Date**: The date when the transaction took place, treated as interval data because the difference between dates is meaningful.

Random Data Generation and Sensible Ranges:

- **TransactionID**: Randomly generated within the range of 1000000 to 9999999, ensuring uniqueness.
- **CustomerID**: Selected from the list of customer IDs in the 'Accounts' table, ensuring that transactions are linked to actual customers.

- **Transaction Type**: Randomly chosen from a predefined list of transaction types (Deposit, Withdrawal, Payment, Transfer), which are typical transaction categories in banking.

- **Amount**: Randomly generated to be between 10 and 10,000, with two decimal places. This range encompasses typical transaction amounts, from small to significant sums.

- **Date**: Randomly generated to fall between the years 2000 and 2022, which are plausible dates for banking transactions.

- **DataFrame Structure:** The DataFrame, `transactions shape`, is expected to have 1000 rows for transactions and 5 columns for the attributes mentioned above.

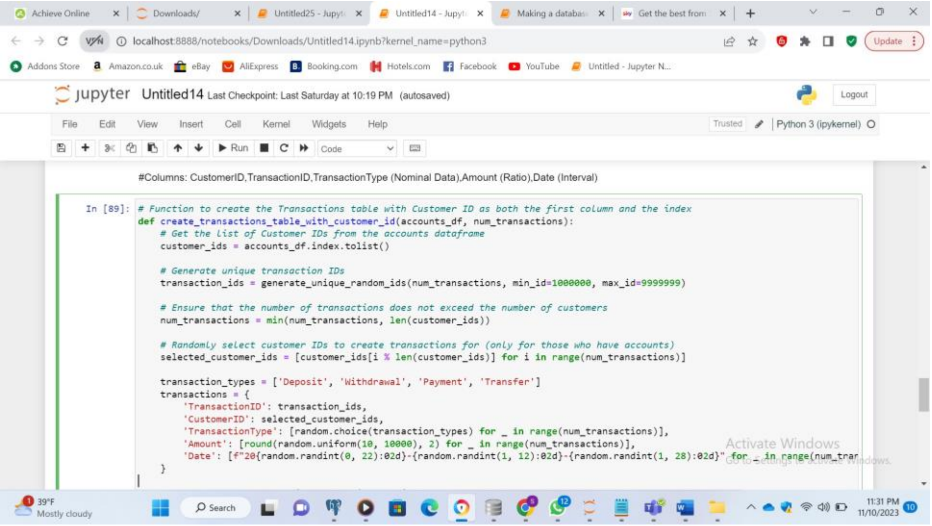
- **CustomerID**: Selected from the list of customer IDs in the 'Accounts' table, ensuring that transactions are linked to actual customers.

- **Transaction Type**: Randomly chosen from a predefined list of transaction types (Deposit, Withdrawal, Payment, Transfer), which are typical transaction categories in banking.

- **Amount**: Randomly generated to be between 10 and 10,000, with two decimal places. This range encompasses typical transaction amounts, from small to significant sums.

- **Date**: Randomly generated to fall between the years 2000 and 2022, which are plausible dates for banking transactions.

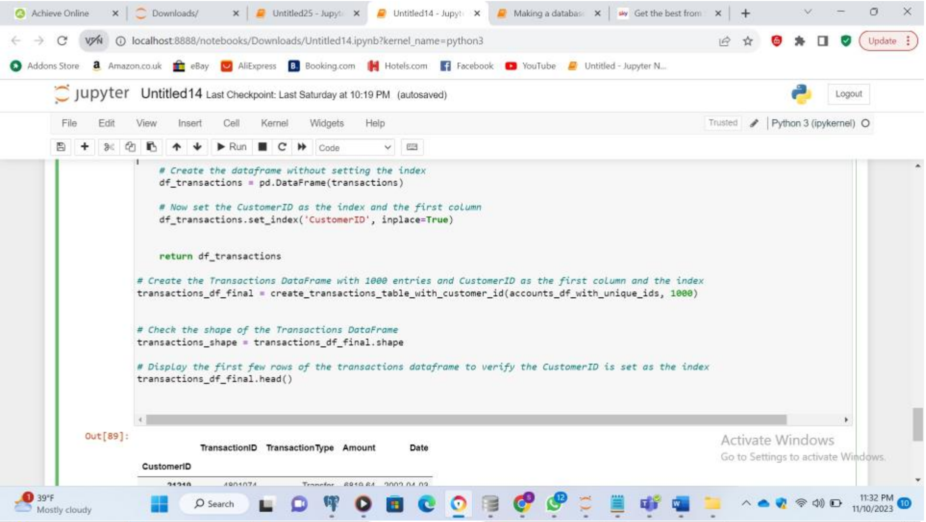
"**Sample Code Blocks:** Transaction table


jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

#Columns: CustomerID,TransactionID,TransactionType (Nominal Data),Amount (Ratio),Date (Interval)

In [89]:

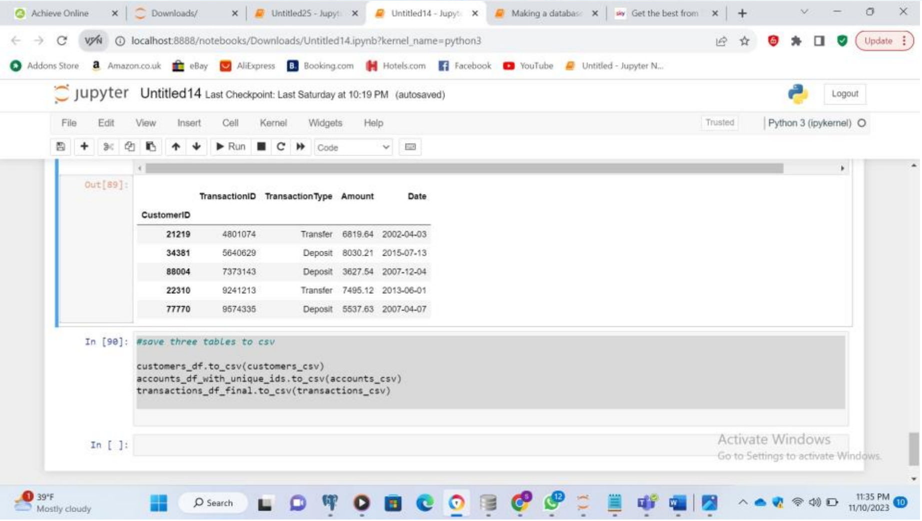
```
# Function to create the Transactions table with Customer ID as both the first column and the index
def create_transactions_table_with_customer_id(accounts_df, num_transactions):
    # Get the list of Customer IDs from the accounts dataframe
    customer_ids = accounts_df.index.tolist()

    # Generate unique transaction IDs
    transaction_ids = generate_unique_random_ids(num_transactions, min_id=1000000, max_id=9999999)

    # Ensure that the number of transactions does not exceed the number of customers
    num_transactions = min(num_transactions, len(customer_ids))

    # Randomly select customer IDs to create transactions for (only for those who have accounts)
    selected_customer_ids = [customer_ids[i % len(customer_ids)] for i in range(num_transactions)]

    transaction_types = ['Deposit', 'Withdrawal', 'Payment', 'Transfer']
    transactions = {
        'TransactionID': transaction_ids,
        'CustomerID': selected_customer_ids,
        'TransactionType': [random.choice(transaction_types) for _ in range(num_transactions)],
        'Amount': [round(random.uniform(10, 10000), 2) for _ in range(num_transactions)],
        'Date': [f"20{random.randint(0, 22):02d}-{random.randint(1, 12):02d}-{random.randint(1, 28):02d}" for _ in range(num_trar
    }
```


jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

```
    # Create the dataframe without setting the index
    df_transactions = pd.DataFrame(transactions)

    # Now set the CustomerID as the index and the first column
    df_transactions.set_index('CustomerID', inplace=True)


    return df_transactions

# Create the Transactions DataFrame with 1000 entries and CustomerID as the first column and the index
transactions_df_final = create_transactions_table_with_customer_id(accounts_df_with_unique_ids, 1000)


# Check the shape of the Transactions DataFrame
transactions_shape = transactions_df_final.shape

# Display the first few rows of the transactions dataframe to verify the CustomerID is set as the index
transactions_df_final.head()
```

Out[89]:

TransactionID TransactionType Amount Date
CustomerID


jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

Out[89]:

CustomerID	TransactionID	TransactionType	Amount	Date
21219	4801074	Transfer	6819.64	2002-04-03
34381	5640629	Deposit	8030.21	2015-07-13
88004	7373143	Deposit	3627.54	2007-12-04
22310	9241213	Transfer	7495.12	2013-06-01
77770	9574335	Deposit	5537.63	2007-04-07

In [90]:

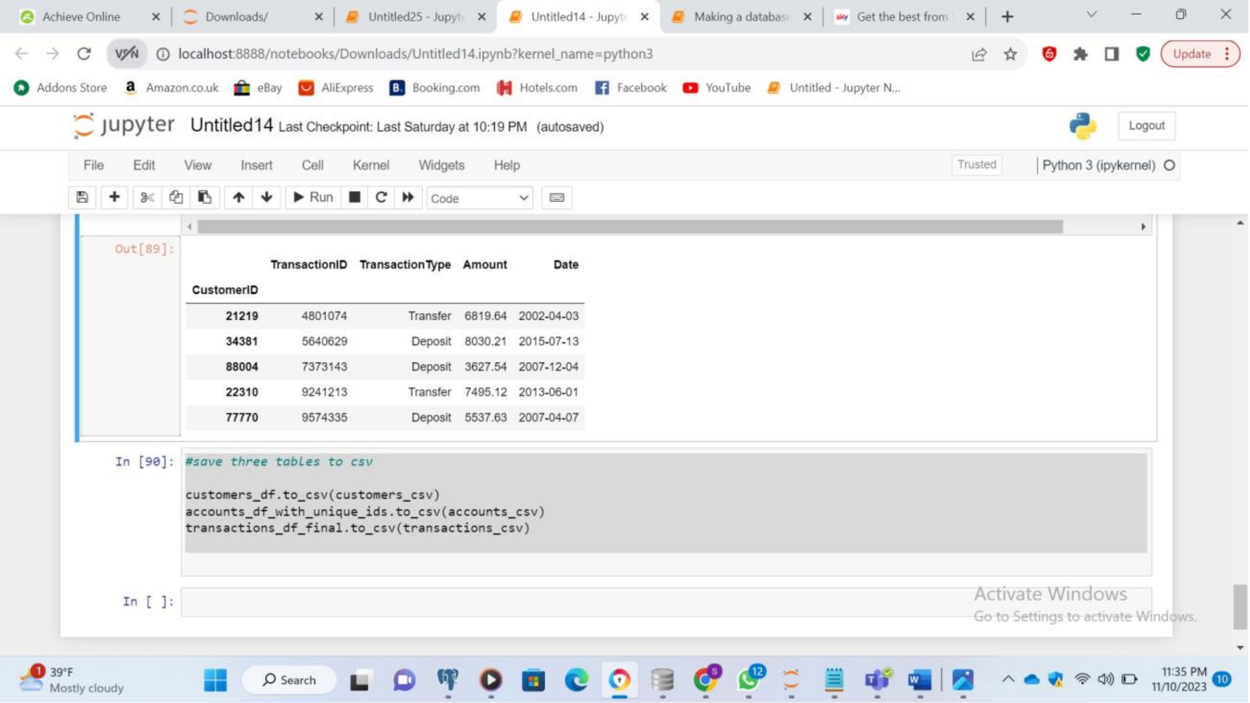
```
#save three tables to csv

customers_df.to_csv(customers_csv)
accounts_df_with_unique_ids.to_csv(accounts_csv)
transactions_df_final.to_csv(transactions_csv)
```

In [ ]:

Data Processing Workflow

Upon the successful creation of the tables, they are saved in a CSV (Comma-Separated Values) format. Subsequently, reports generated from these tables are prepared in Microsoft Excel, which are then imported into an SQLite database. This facilitates further querying and analysis of the data.

"Sample Code Blocks:



```
Out[89]:
            TransactionID  TransactionType  Amount        Date
CustomerID
     21219        4801074         Transfer  6819.64  2002-04-03
     34381        5640629          Deposit  8030.21  2015-07-13
     88004        7373143          Deposit  3627.54  2007-12-04
     22310        9241213         Transfer  7495.12  2013-06-01
     77770        9574335          Deposit  5537.63  2007-04-07
```

```
In [90]: #save three tables to csv

         customers_df.to_csv(customers_csv)
         accounts_df_with_unique_ids.to_csv(accounts_csv)
         transactions_df_final.to_csv(transactions_csv)

In [ ]:
```

localhost:8888/notebooks/Downloads/Untitled14.ipynb?kernel_name=python3
jupyter Untitled14 Last Checkpoint: Last Saturday at 10:19 PM (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3 (ipykernel)
Run
Code
Activate Windows
Go to Settings to activate Windows.
39°F
Mostly cloudy
Search
11:35 PM
11/10/2023

DATABASE SCHEMA

In the context, the schema for the multi-table database consists of three interconnected tables: 'Customers', 'Accounts', and 'Transactions'. Each table has attributes with constraints to ensure data integrity and sensible data ranges. Here is the detailed schema for each table:

Database Schema Overview

Customers Table:

- **`CustomerID`**: Primary key, integer (5-digit).
- **`Name`**: String, full name.
- **`Age`**: Integer, between 18 and 70.
- **`Gender`**: String, 'Male', 'Female', or 'Other'.
- **`Email`**: String, email format (optional).

Account Table:

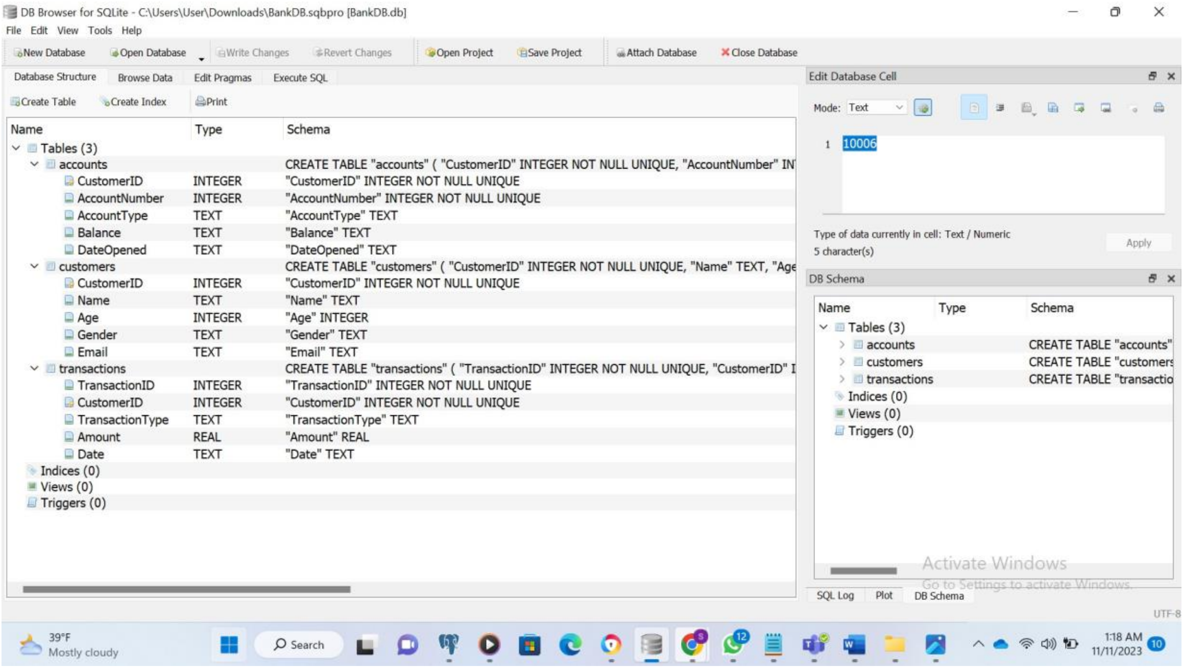
- **`AccountNumber`**: Primary key, integer (8-digit).
- **`CustomerID`**: Foreign key, integer, references **`Customers`**.
- **`AccountType`**: String, 'Basic', 'Standard', or 'Premium'.
- **`Balance`**: Float, positive values.
- **`DateOpened`**: Date, from 2000 to present.

Transaction Tables:

- **`TransactionID`**: Primary key, integer (7-digit).
- **`CustomerID`**: Foreign key, integer, references **`Customers`**.
- **`TransactionType`**: String, 'Deposit', 'Withdrawal', 'Payment', 'Transfer'.
- **`Amount`**: Float
- **`Date`**: Date, within the same range as **`DateOpened`**.
-

Note: Each table has a unique primary key **`CustomerID`** for 'Customers', **`AccountNumber`** for 'Accounts', and **`TransactionID`** for 'Transactions'. **`CustomerID`** also serves as a foreign key in the 'Accounts' and 'Transactions' tables. Should uniqueness not be maintained by these identifiers alone, composite keys would be implemented.

Visual Guide to SQLite Database Schema Structure



Name	Type	Schema
Tables (3)		
accounts		CREATE TABLE "accounts" ("CustomerID" INTEGER NOT NULL UNIQUE, "AccountNumber" IN
CustomerID	INTEGER	"CustomerID" INTEGER NOT NULL UNIQUE
AccountNumber	INTEGER	"AccountNumber" INTEGER NOT NULL UNIQUE
AccountType	TEXT	"AccountType" TEXT
Balance	TEXT	"Balance" TEXT
DateOpened	TEXT	"DateOpened" TEXT
customers		CREATE TABLE "customers" ("CustomerID" INTEGER NOT NULL UNIQUE, "Name" TEXT, "Age
CustomerID	INTEGER	"CustomerID" INTEGER NOT NULL UNIQUE
Name	TEXT	"Name" TEXT
Age	INTEGER	"Age" INTEGER
Gender	TEXT	"Gender" TEXT
Email	TEXT	"Email" TEXT
transactions		CREATE TABLE "transactions" ("TransactionID" INTEGER NOT NULL UNIQUE, "CustomerID" I
TransactionID	INTEGER	"TransactionID" INTEGER NOT NULL UNIQUE
CustomerID	INTEGER	"CustomerID" INTEGER NOT NULL UNIQUE
TransactionType	TEXT	"TransactionType" TEXT
Amount	REAL	"Amount" REAL
Date	TEXT	"Date" TEXT
Indices (0)		
Views (0)		
Triggers (0)		

Mode: Text

```
1 10006
```

Type of data currently in cell: Text / Numeric
5 character(s)

Name	Type	Schema
Tables (3)		
accounts		CREATE TABLE "accounts"
customers		CREATE TABLE "customers
transactions		CREATE TABLE "transactio
Indices (0)		
Views (0)		
Triggers (0)		

Justification for Separate Tables:

- **Normalization**: The 'Customers', 'Accounts', and 'Transactions' tables follow database normalization to reduce redundancy and improve data integrity.

- **Maintenance**: It's easier to manage and update data when it's organized into logical groupings.
- **Performance**: Querying smaller, related tables is more efficient than querying a single large table.
- **Security**: Different data sensitivity levels can be managed with distinct access controls for each table.

Ethical Discussion and Data Privacy:

The data generated is synthetic, ensuring no real personal data is used, which aligns with ethical standards:

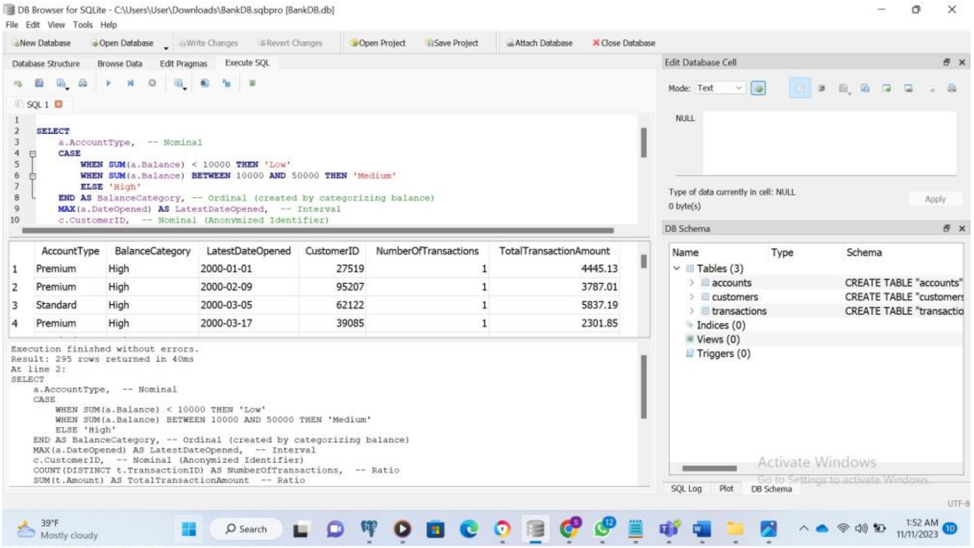
- **Anonymity**: Randomly generated data ensures no real individuals can be identified.
- **Data Minimization**: Only essential data for analysis is generated, avoiding excessive personal details.
- **Compliance**: Practices mimic compliance with data protection laws like GDPR, demonstrating good data governance.

Note: In a real-world application, sensitive data would be encrypted, and access would be strictly controlled. The code here provides a model for educational purposes without actual.

Example queries of your database including joins and selections

The given SQL query exemplifies advanced data retrieval using joins across three tables: 'Customers', 'Accounts', and 'Transactions'. It showcases the use of SQL's capabilities to handle complex data types and table relationships.

The SQL query combines 'Customers', 'Accounts', and 'Transactions' tables, using INNER and LEFT OUTER JOINs to reflect relationships and ensure all relevant data is included. It employs a CASE statement to categorize balance sums and uses aggregate functions like SUM, MAX, and COUNT to summarize data. The results are grouped by 'AccountType' and 'CustomerID' and filtered with a HAVING clause for balances over 70,000. Finally, the query orders the output by the latest 'DateOpened'.



```

SELECT
    a.AccountType,  -- Nominal
    CASE
        WHEN SUM(a.Balance) < 10000 THEN 'Low'
        WHEN SUM(a.Balance) BETWEEN 10000 AND 50000 THEN 'Medium'
        ELSE 'High'
    END AS BalanceCategory, -- Ordinal (created by categorizing balance)
    MAX(a.DateOpened) AS LatestDateOpened,  -- Interval
    c.CustomerID,  -- Nominal (Anonymized Identifier)
```

	AccountType	BalanceCategory	LatestDateOpened	CustomerID	NumberOfTransactions	TotalTransactionAmount
1	Premium	High	2000-01-01	27519	1	4445.13
2	Premium	High	2000-02-09	95207	1	3787.01
3	Standard	High	2000-03-05	62122	1	5837.19
4	Premium	High	2000-03-17	39085	1	2301.85

```
Execution finished without errors.
Result: 295 rows returned in 40ms
At line 2:
SELECT
    a.AccountType,  -- Nominal
    CASE
        WHEN SUM(a.Balance) < 10000 THEN 'Low'
        WHEN SUM(a.Balance) BETWEEN 10000 AND 50000 THEN 'Medium'
        ELSE 'High'
    END AS BalanceCategory, -- Ordinal (created by categorizing balance)
    MAX(a.DateOpened) AS LatestDateOpened,  -- Interval
    c.CustomerID,  -- Nominal (Anonymized Identifier)
    COUNT(DISTINCT t.TransactionID) AS NumberOfTransactions,  -- Ratio
    SUM(t.Amount) AS TotalTransactionAmount  -- Ratio
```

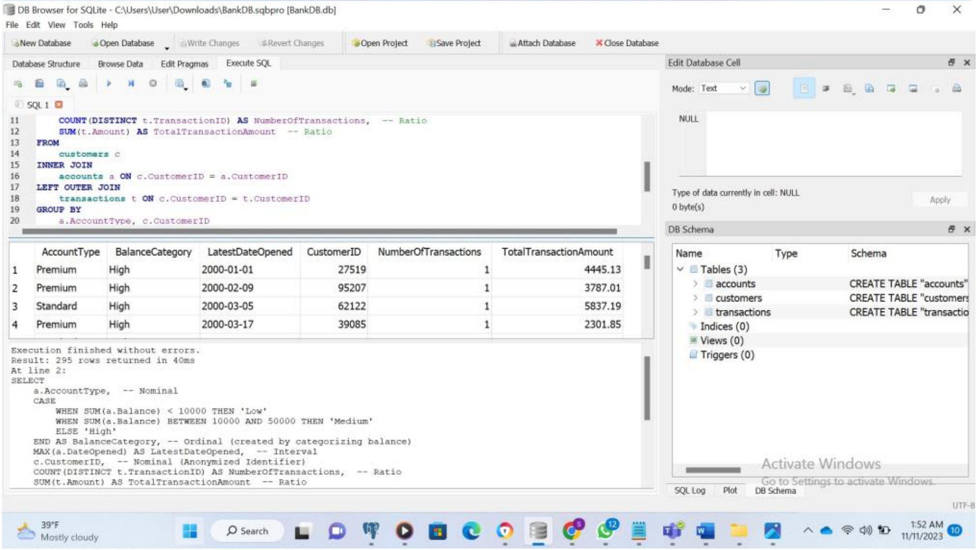
Mode: Text

NULL

Type of data currently in cell: NULL
0 byte(s)

Apply

Name	Type	Schema
Tables (3)		
accounts		CREATE TABLE "accounts"
customers		CREATE TABLE "customers
transactions		CREATE TABLE "transactio
Indices (0)		
Views (0)		
Triggers (0)		



```
    COUNT(DISTINCT t.TransactionID) AS NumberOfTransactions,  -- Ratio
    SUM(t.Amount) AS TotalTransactionAmount  -- Ratio
FROM
    customers c
INNER JOIN
    accounts a ON c.CustomerID = a.CustomerID
LEFT OUTER JOIN
    transactions t ON c.CustomerID = t.CustomerID
GROUP BY
    a.AccountType, c.CustomerID
```

	AccountType	BalanceCategory	LatestDateOpened	CustomerID	NumberOfTransactions	TotalTransactionAmount
1	Premium	High	2000-01-01	27519	1	4445.13
2	Premium	High	2000-02-09	95207	1	3787.01
3	Standard	High	2000-03-05	62122	1	5837.19
4	Premium	High	2000-03-17	39085	1	2301.85

```
Execution finished without errors.
Result: 295 rows returned in 40ms
At line 2:
SELECT
    a.AccountType,  -- Nominal
    CASE
        WHEN SUM(a.Balance) < 10000 THEN 'Low'
        WHEN SUM(a.Balance) BETWEEN 10000 AND 50000 THEN 'Medium'
        ELSE 'High'
    END AS BalanceCategory, -- Ordinal (created by categorizing balance)
    MAX(a.DateOpened) AS LatestDateOpened,  -- Interval
    c.CustomerID,  -- Nominal (Anonymized Identifier)
    COUNT(DISTINCT t.TransactionID) AS NumberOfTransactions,  -- Ratio
    SUM(t.Amount) AS TotalTransactionAmount  -- Ratio
```

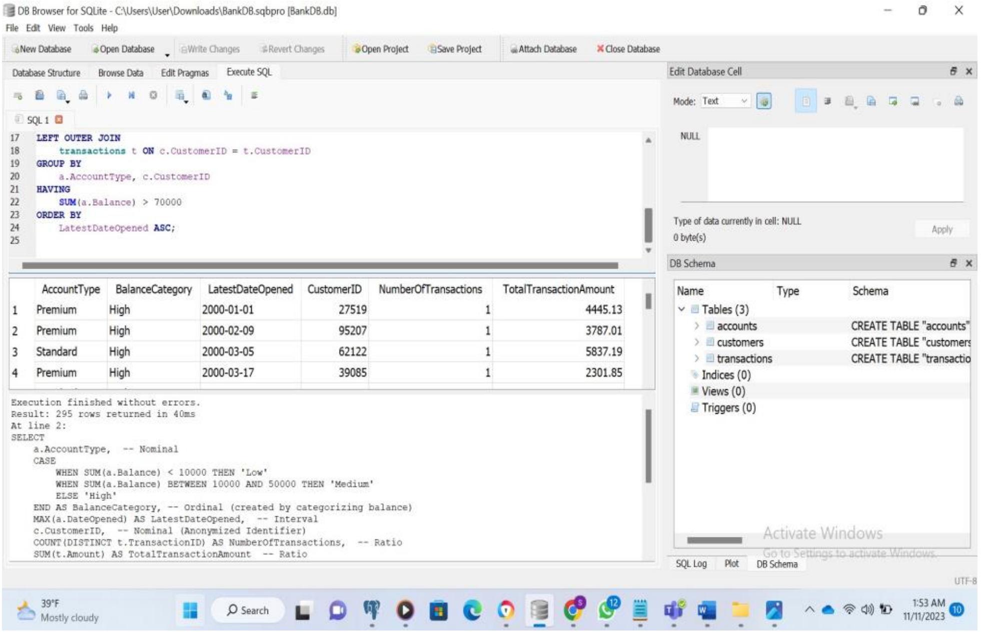
Mode: Text

NULL

Type of data currently in cell: NULL
0 byte(s)

Apply

Name	Type	Schema
Tables (3)		
accounts		CREATE TABLE "accounts"
customers		CREATE TABLE "customers
transactions		CREATE TABLE "transactio
Indices (0)		
Views (0)		
Triggers (0)		



```
LEFT OUTER JOIN
    transactions t ON c.CustomerID = t.CustomerID
GROUP BY
    a.AccountType, c.CustomerID
HAVING
    SUM(a.Balance) > 70000
ORDER BY
    LatestDateOpened ASC;

```

	AccountType	BalanceCategory	LatestDateOpened	CustomerID	NumberOfTransactions	TotalTransactionAmount
1	Premium	High	2000-01-01	27519	1	4445.13
2	Premium	High	2000-02-09	95207	1	3787.01
3	Standard	High	2000-03-05	62122	1	5837.19
4	Premium	High	2000-03-17	39085	1	2301.85

```
Execution finished without errors.
Result: 295 rows returned in 40ms
At line 2:
SELECT
    a.AccountType,  -- Nominal
    CASE
        WHEN SUM(a.Balance) < 10000 THEN 'Low'
        WHEN SUM(a.Balance) BETWEEN 10000 AND 50000 THEN 'Medium'
        ELSE 'High'
    END AS BalanceCategory, -- Ordinal (created by categorizing balance)
    MAX(a.DateOpened) AS LatestDateOpened,  -- Interval
    c.CustomerID,  -- Nominal (Anonymized Identifier)
    COUNT(DISTINCT t.TransactionID) AS NumberOfTransactions,  -- Ratio
    SUM(t.Amount) AS TotalTransactionAmount  -- Ratio
```

Mode: Text

NULL

Type of data currently in cell: NULL
0 byte(s)

Apply

Name	Type	Schema
Tables (3)		
accounts		CREATE TABLE "accounts"
customers		CREATE TABLE "customers
transactions		CREATE TABLE "transactio
Indices (0)		
Views (0)		
Triggers (0)		

- **JOIN Clauses**: The **INNER JOIN** between 'customers' and 'accounts' ensures that only customers with accounts are included. The **LEFT OUTER JOIN** with 'transactions' includes all customers and their accounts, even if they have no transactions, indicating a one-to-many relationship between customers and transactions.

- **CASE Statement**: This conditional logic creates an ordinal data type category based on the sum of account balances, classifying them as 'Low', 'Medium', or 'High'.

- **Aggregate Functions**: **SUM**, **MAX**, and **COUNT(DISTINCT)** are used to calculate total balances, the latest date an account was opened, and the number of unique transactions, respectively. These illustrate the ratio and interval data types through summarization of transaction amounts and dates.

- **GROUP BY Clause**: Groups the results by 'AccountType' and 'CustomerID', which is necessary for the aggregate functions to calculate values per group.

- **HAVING Clause**: Filters the groups to include only those where the total balance sum exceeds 70,000. This is an example of using aggregate functions in conditions post-grouping.

- **ORDER BY Clause**: Orders the results by the most recent 'DateOpened', showcasing sorting on interval data.

NOTE: The query generates a report detailing customer accounts by balance tiers, recent account activity, transaction count, and total transaction value, specifically for high-balance accounts. It showcases SQL's analytical utility across multiple tables, using anonymized identifiers such as Customer ID to maintain privacy.

Conclusion

The report demonstrates the creation of a synthetic financial database schema with 'Customers', 'Accounts', and 'Transactions' tables, illustrating sound database normalization. It promotes data integrity and efficient querying while incorporating security measures for data privacy. The synthetic nature of the dataset maintains anonymity, aligning with data protection regulations. Additionally, the example SQL query highlights skillful data retrieval and analysis techniques, using SQL's capabilities to join tables, compute aggregates, and categorize data, all while emphasizing data privacy ethics. This schema and its queries not only serve as instructional material for database and SQL education but also underscore the necessity of responsible data management in practice.