Data Engineering

SQL Coding Challenge

J Jatin

Database Design and Table Data:

-- Database Design

```
USE CodingChallenge;

GO
```

-- Create Customers Table

```
CREATE TABLE Customers (
   customer_id INT IDENTITY(1,1) PRIMARY KEY,
   first_name NVARCHAR(50) NOT NULL,
   last_name NVARCHAR(50) NOT NULL,
   DOB DATE NOT NULL,
   email NVARCHAR(100) NOT NULL UNIQUE,
   phone_number NVARCHAR(15) NOT NULL,
   address NVARCHAR(255)
);
```

-- Create Accounts Table

```
CREATE TABLE Accounts (
   account_id INT IDENTITY(1,1) PRIMARY KEY,
   customer_id INT NOT NULL,
   account_type NVARCHAR(20) CHECK (account_type IN ('savings',
'current', 'zero_balance')),
   balance DECIMAL(10, 2) NOT NULL,
   FOREIGN KEY (customer_id) REFERENCES
Customers(customer_id) ON DELETE CASCADE
);
```

-- Create Transactions Table

```
CREATE TABLE Transactions (
    transaction_id INT IDENTITY(1,1) PRIMARY KEY,
    account_id INT NOT NULL,
    transaction_type NVARCHAR(20) CHECK (transaction_type IN ('deposit', 'withdrawal', 'transfer')),
    amount DECIMAL(10, 2) NOT NULL,
    transaction_date DATETIME DEFAULT GETDATE(),
    FOREIGN KEY (account_id) REFERENCES Accounts(account_id) ON DELETE CASCADE
);
```

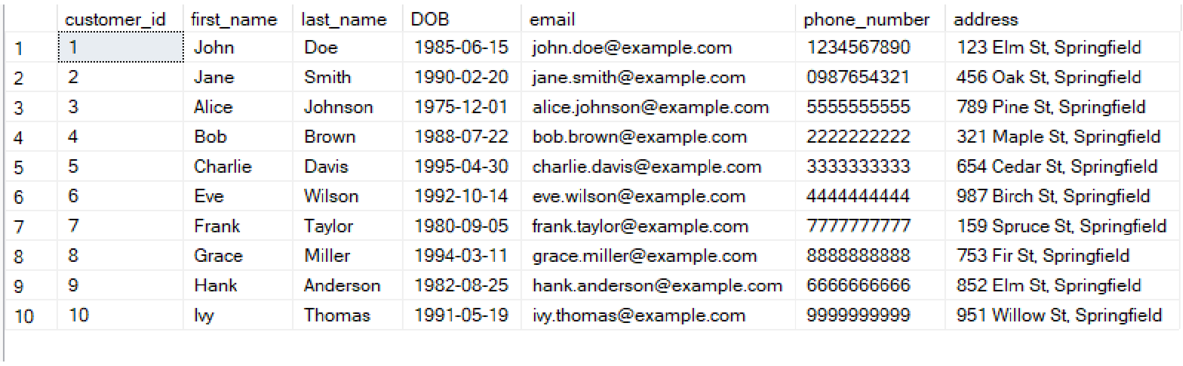
Insert 10 sample records into each of the following tables.

--Customers

--Accounts

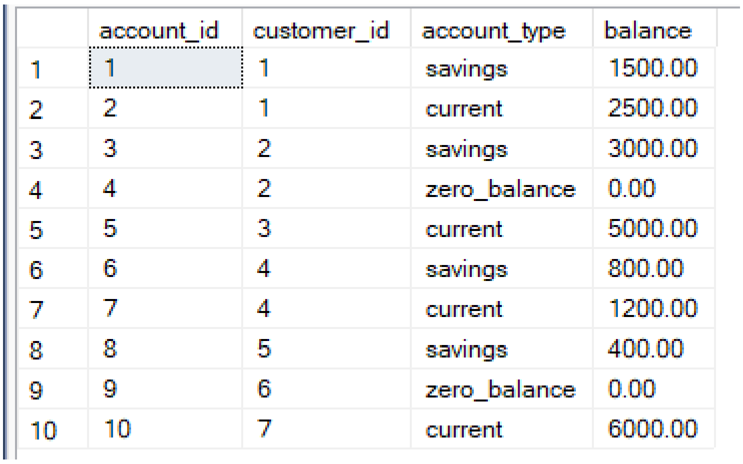
--Transactions

```
INSERT INTO Customers (first_name, last_name, DOB, email, phone_number, address)
VALUES
('John', 'Doe', '1985-06-15', 'john.doe@example.com', '1234567890', '123 Elm St, Springfield'),
('Jane', 'Smith', '1990-02-20', 'jane.smith@example.com', '0987654321', '456 Oak St, Springfield'),
('Alice', 'Johnson', '1975-12-01', 'alice.johnson@example.com', '5555555555', '789 Pine St, Springfield'),
('Bob', 'Brown', '1988-07-22', 'bob.brown@example.com', '2222222222', '321 Maple St, Springfield'),

('Charlie', 'Davis', '1995-04-30', 'charlie.davis@example.com', '3333333333', '654 Cedar St, Springfield'),

('Eve', 'Wilson', '1992-10-14', 'eve.wilson@example.com', '4444444444', '987 Birch St, Springfield'),

('Frank', 'Taylor', '1980-09-05', 'frank.taylor@example.com', '7777777777', '159 Spruce St, Springfield'),

('Grace', 'Miller', '1994-03-11', 'grace.miller@example.com', '8888888888', '753 Fir St, Springfield'),

('Hank', 'Anderson', '1982-08-25', 'hank.anderson@example.com', '6666666666', '852 Elm St, Springfield'),

('Ivy', 'Thomas', '1991-05-19', 'ivy.thomas@example.com', '9999999999', '951 Willow St, Springfield');
```



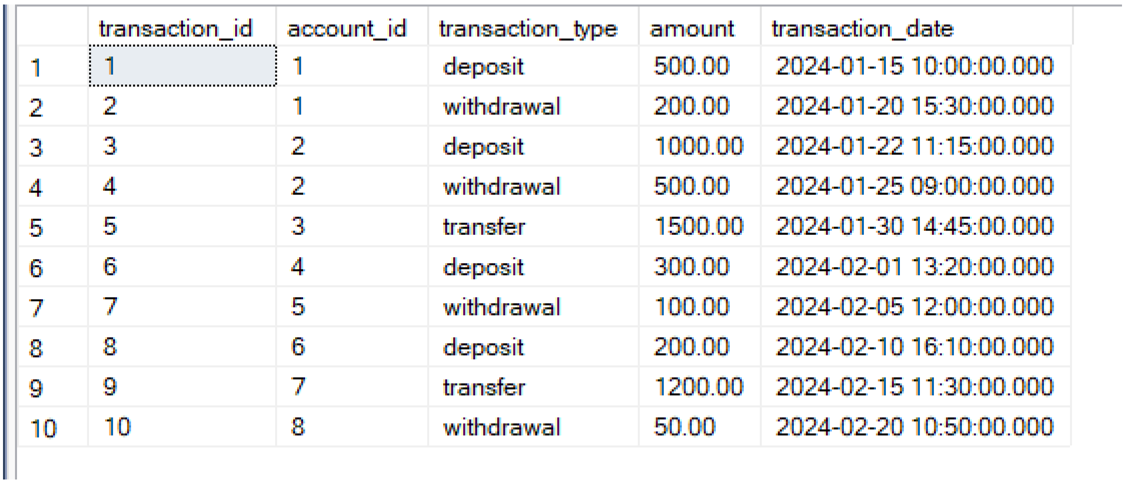
	customer_id	first_name	last_name	DOB	email	phone_number	address
1	1	John	Doe	1985-06-15	john.doe@example.com	1234567890	123 Elm St, Springfield
2	2	Jane	Smith	1990-02-20	jane.smith@example.com	0987654321	456 Oak St, Springfield
3	3	Alice	Johnson	1975-12-01	alice.johnson@example.com	5555555555	789 Pine St, Springfield
4	4	Bob	Brown	1988-07-22	bob.brown@example.com	2222222222	321 Maple St, Springfield
5	5	Charlie	Davis	1995-04-30	charlie.davis@example.com	3333333333	654 Cedar St, Springfield
6	6	Eve	Wilson	1992-10-14	eve.wilson@example.com	4444444444	987 Birch St, Springfield
7	7	Frank	Taylor	1980-09-05	frank.taylor@example.com	7777777777	159 Spruce St, Springfield
8	8	Grace	Miller	1994-03-11	grace.miller@example.com	8888888888	753 Fir St, Springfield
9	9	Hank	Anderson	1982-08-25	hank.anderson@example.com	6666666666	852 Elm St, Springfield
10	10	Ivy	Thomas	1991-05-19	ivy.thomas@example.com	9999999999	951 Willow St, Springfield

```
INSERT INTO Accounts (customer_id, account_type, balance)

VALUES

(1, 'savings', 1500.00),

(1, 'current', 2500.00),

(2, 'savings', 3000.00),

(2, 'zero_balance', 0.00),

(3, 'current', 5000.00),

(4, 'savings', 800.00),

(4, 'current', 1200.00),
(5, 'savings', 400.00),

(6, 'zero_balance', 0.00),

(7, 'current', 6000.00);
```



	account_id	customer_id	account_type	balance
1	1	1	savings	1500.00
2	2	1	current	2500.00
3	3	2	savings	3000.00
4	4	2	zero_balance	0.00
5	5	3	current	5000.00
6	6	4	savings	800.00
7	7	4	current	1200.00
8	8	5	savings	400.00
9	9	6	zero_balance	0.00
10	10	7	current	6000.00

```
INSERT INTO Transactions (account_id, transaction_type, amount, transaction_date)

VALUES

(1, 'deposit', 500.00, '2024-01-15 10:00:00'),

(1, 'withdrawal', 200.00, '2024-01-20 15:30:00'),

(2, 'deposit', 1000.00, '2024-01-22 11:15:00'),

(2, 'withdrawal', 500.00, '2024-01-25 09:00:00'),

(3, 'transfer', 1500.00, '2024-01-30 14:45:00'),

(4, 'deposit', 300.00, '2024-02-01 13:20:00'),

(5, 'withdrawal', 100.00, '2024-02-05 12:00:00'),

(6, 'deposit', 200.00, '2024-02-10 16:10:00'),

(7, 'transfer', 1200.00, '2024-02-15 11:30:00'),

(8, 'withdrawal', 50.00, '2024-02-20 10:50:00');
```



	transaction_id	account_id	transaction_type	amount	transaction_date
1	1	1	deposit	500.00	2024-01-15 10:00:00.000
2	2	1	withdrawal	200.00	2024-01-20 15:30:00.000
3	3	2	deposit	1000.00	2024-01-22 11:15:00.000
4	4	2	withdrawal	500.00	2024-01-25 09:00:00.000
5	5	3	transfer	1500.00	2024-01-30 14:45:00.000
6	6	4	deposit	300.00	2024-02-01 13:20:00.000
7	7	5	withdrawal	100.00	2024-02-05 12:00:00.000
8	8	6	deposit	200.00	2024-02-10 16:10:00.000
9	9	7	transfer	1200.00	2024-02-15 11:30:00.000
10	10	8	withdrawal	50.00	2024-02-20 10:50:00.000

About Joins:

Joins are used in SQL to combine rows from two or more tables based on a related column between them. The purpose of a join is to create a meaningful dataset by bringing together data from different sources, typically by matching values in common columns.

There are different types of joins in SQL:

- **INNER JOIN:** Returns only the rows where there is a match in both tables.
- **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table, and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.
- **RIGHT JOIN (or RIGHT OUTER JOIN):** Similar to the left join but returns all rows from the right table and the matched rows from the left table.
- **FULL JOIN (or FULL OUTER JOIN):** Returns all rows when there is a match in either table. If there is no match, NULL values are returned for non-matching rows.

About Sub-Queries:

A **subquery (or inner query)** is a query within another SQL query. The result of the subquery is used by the outer query to further refine the result set. Subqueries can be used in SELECT, INSERT, UPDATE, and DELETE statements.

Subqueries are often used for:

- Retrieving data that will be used in the main query as a condition.
- Nesting queries to break down complex logic into smaller, more manageable pieces.

There are two main types of subqueries:

- Correlated Subquery: The subquery depends on the outer query for its values.
- Non-correlated Subquery: The subquery is independent and can be executed on its own.

About Sub-Total:

A **subtotal** in SQL can be calculated using the **ROLLUP** function, which is part of the GROUP BY clause. **ROLLUP** allows you to calculate aggregates (like sums or counts) and include additional rows that show subtotals and grand totals.

- **Subtotal** refers to the aggregate value for a particular group of data (e.g., total balance for each account type).
- **Grand Total** refers to the sum or total across all groups combined.

The ROLLUP function is an extension of GROUP BY and generates subtotals automatically.

Queries Outputs:

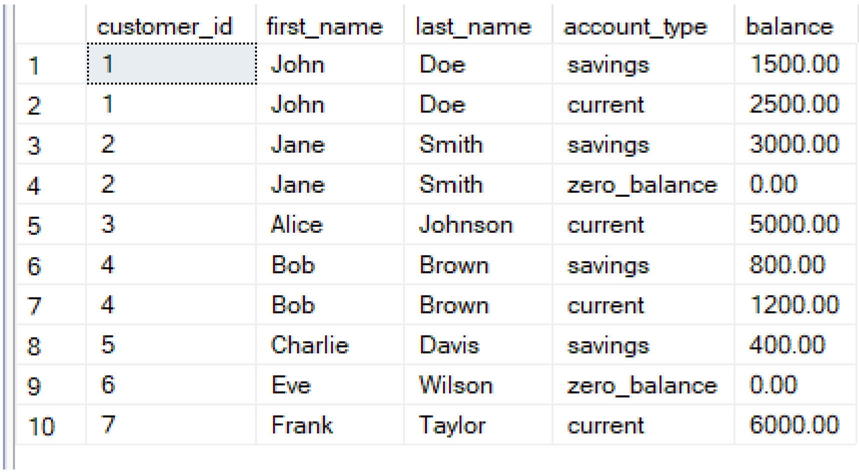
--Querying Data by Using Joins and Subqueries and Subtotal (Rollup in MSSQL)

--Query 1: List all customers along with their account details using an INNER JOIN.

SELECT C.customer_id, C.first_name, C.last_name, A.account_type, A.balance

FROM Customers C

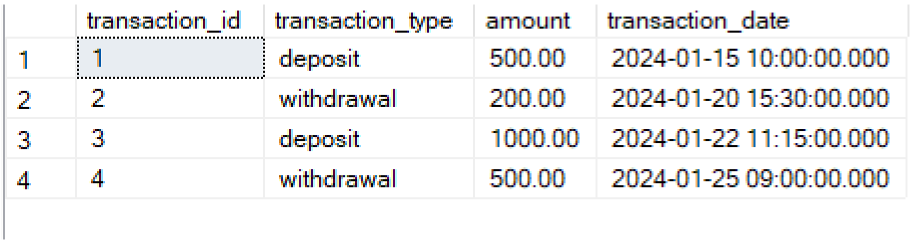
INNER JOIN Accounts A ON C.customer_id = A.customer_id;



	customer_id	first_name	last_name	account_type	balance
1	1	John	Doe	savings	1500.00
2	1	John	Doe	current	2500.00
3	2	Jane	Smith	savings	3000.00
4	2	Jane	Smith	zero_balance	0.00
5	3	Alice	Johnson	current	5000.00
6	4	Bob	Brown	savings	800.00
7	4	Bob	Brown	current	1200.00
8	5	Charlie	Davis	savings	400.00
9	6	Eve	Wilson	zero_balance	0.00
10	7	Frank	Taylor	current	6000.00

--Query 2: Retrieve the transaction history of a specific customer (e.g., John Doe) using a subquery.

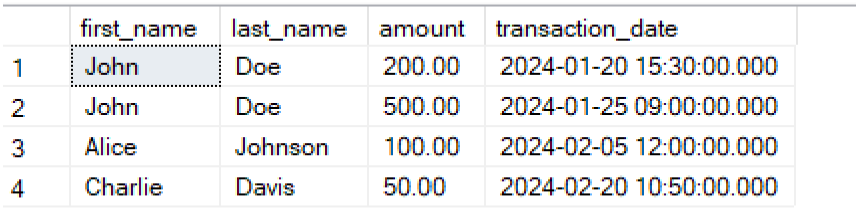
```
SELECT T.transaction_id, T.transaction_type, T.amount,
T.transaction_date

FROM Transactions T

WHERE T.account_id IN (

   SELECT A.account_id

   FROM Accounts A

   JOIN Customers C ON A.customer_id = C.customer_id

   WHERE C.first_name = 'John' AND C.last_name = 'Doe'

);
```



	transaction_id	transaction_type	amount	transaction_date
1	1	deposit	500.00	2024-01-15 10:00:00.000
2	2	withdrawal	200.00	2024-01-20 15:30:00.000
3	3	deposit	1000.00	2024-01-22 11:15:00.000
4	4	withdrawal	500.00	2024-01-25 09:00:00.000

-- Query 3: Find customers who have made withdrawals and display the amount withdrawn using a JOIN.

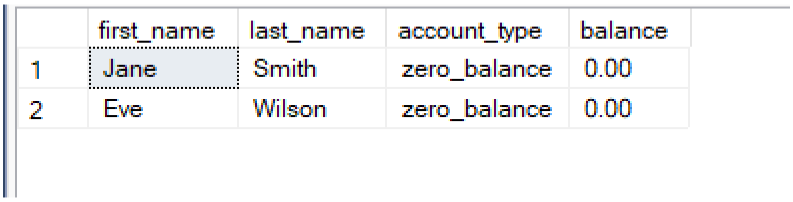
```
SELECT C.first_name, C.last_name, T.amount, T.transaction_date

FROM Customers C

JOIN Accounts A ON C.customer_id = A.customer_id
JOIN Transactions T ON A.account_id = T.account_id

WHERE T.transaction_type = 'withdrawal';
```



	first_name	last_name	amount	transaction_date
1	John	Doe	200.00	2024-01-20 15:30:00.000
2	John	Doe	500.00	2024-01-25 09:00:00.000
3	Alice	Johnson	100.00	2024-02-05 12:00:00.000
4	Charlie	Davis	50.00	2024-02-20 10:50:00.000

--Query 4: List customers who have zero balance accounts using a JOIN.

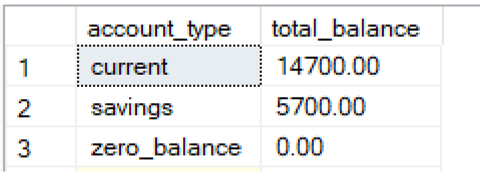
```
SELECT C.first_name, C.last_name, A.account_type, A.balance

FROM Customers C

JOIN Accounts A ON C.customer_id = A.customer_id

WHERE A.balance = 0;
```



	first_name	last_name	account_type	balance
1	Jane	Smith	zero_balance	0.00
2	Eve	Wilson	zero_balance	0.00

--Query 5: Calculate the total balance for each account type using Subtotal (Rollup in MSSQL).

```
SELECT

    account_type,

    SUM(balance) AS total_balance

FROM

    Accounts

GROUP BY

    ROLLUP(account_type);
```



	account_type	total_balance
1	current	14700.00
2	savings	5700.00
3	zero_balance	0.00

About Group-By:

The **GROUP BY** clause in SQL is used to arrange identical data into groups. It is typically used with aggregate functions (like SUM(), COUNT(), AVG(), MAX(), MIN()) to perform calculations on each group of data.

Purpose:

- To aggregate data into categories or groups and apply calculations (e.g., finding the total balance for each account type).
- It helps summarize large datasets by grouping rows based on a common column value.

About Having Clause:

The **HAVING** clause is used to filter records after the GROUP BY clause has been applied. While the WHERE clause is used to filter records before grouping, HAVING filters records after the groups have been created.

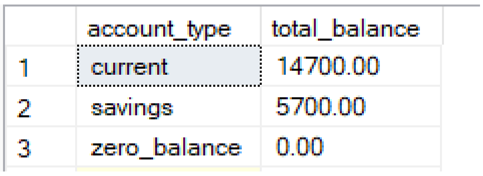
Purpose:

- To filter aggregated data based on conditions applied to the result of an aggregate function.
- It is typically used in conjunction with GROUP BY when we need to restrict the result set based on some condition involving an aggregate function.

Queries Using GROUP BY and HAVING Clauses

--Query 6: Count the number of accounts for each account type using GROUP BY.

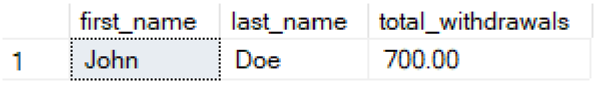
```
SELECT account_type, COUNT(account_id) AS account_count

FROM Accounts

GROUP BY account_type;
```



	account_type	total_balance
1	current	14700.00
2	savings	5700.00
3	zero_balance	0.00

--Query 7: Get the total amount of withdrawals made by each customer where the total amount exceeds 500 using GROUP BY and HAVING.

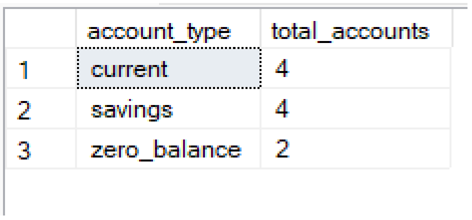
```
SELECT C.first_name, C.last_name, SUM(T.amount) AS
total_withdrawals

FROM Customers C

JOIN Accounts A ON C.customer_id = A.customer_id

JOIN Transactions T ON A.account_id = T.account_id

WHERE T.transaction_type = 'withdrawal'

GROUP BY C.first_name, C.last_name

HAVING SUM(T.amount) > 500;
```



	first_name	last_name	total_withdrawals
1	John	Doe	700.00

--Query 8: Count the Number of Accounts by Account Type and Filter by Having More Than One Account

```
SELECT account_type, COUNT(account_id) AS total_accounts

FROM Accounts

GROUP BY account_type

HAVING COUNT(account_id) > 1;
```



	account_type	total_accounts
1	current	4
2	savings	4
3	zero_balance	2