**SQL CODING CHALLENGE**

**Name: Vikas Reddy Gorantla**

**Date: 08-11-2024**

**Creating Customer Orders Table :**

CREATE TABLE customer\_orders (

order\_id INTEGER NOT NULL,

customer\_id INTEGER NOT NULL,

burger\_id INTEGER NOT NULL,

exclusions VARCHAR(4),

extras VARCHAR(4),

order\_time DATETIME NOT NULL

);

ALTER TABLE customer\_orders

ADD CONSTRAINT fk\_customer\_orders\_burger

FOREIGN KEY (burger\_id) REFERENCES burger\_names(burger\_id);

**Inserting Values In The Table:**

INSERT INTO customer\_orders VALUES (1, 101, 1, NULL, NULL, '2021-01-01 18:05:02');

INSERT INTO customer\_orders VALUES (2, 101, 1, NULL, NULL, '2021-01-01 19:00:52');

INSERT INTO customer\_orders VALUES (3, 102, 1, NULL, NULL, '2021-01-02 23:51:23');

INSERT INTO customer\_orders VALUES (3, 102, 2, NULL, NULL, '2021-01-02 23:51:23');

INSERT INTO customer\_orders VALUES (4, 103, 1, '4', NULL, '2021-01-04 13:23:46');

INSERT INTO customer\_orders VALUES (4, 103, 1, '4', NULL, '2021-01-04 13:23:46');

INSERT INTO customer\_orders VALUES (4, 103, 2, '4', NULL, '2021-01-04 13:23:46');

INSERT INTO customer\_orders VALUES (5, 104, 1, NULL, '1', '2021-01-08 21:00:29');

INSERT INTO customer\_orders VALUES (6, 101, 2, NULL, NULL, '2021-01-08 21:03:13');

INSERT INTO customer\_orders VALUES (7, 105, 2, NULL, '1', '2021-01-08 21:20:29');

INSERT INTO customer\_orders VALUES (8, 102, 1, NULL, NULL, '2021-01-09 23:54:33');

INSERT INTO customer\_orders VALUES (9, 103, 1, '4', '1, 5', '2021-01-10 11:22:59');

INSERT INTO customer\_orders VALUES (10, 104, 1, NULL, NULL, '2021-01-11 18:34:49');

INSERT INTO customer\_orders VALUES (10, 104, 1, '2, 6', '1, 4', '2021-01-11 18:34:49');

**Creating Burger Runner Table:**

CREATE TABLE burger\_runner (

runner\_id INTEGER NOT NULL PRIMARY KEY,

registration\_date DATE NOT NULL

);

**Inserting Values In The Table:**

INSERT INTO burger\_runner VALUES (1, '2021-01-01');

INSERT INTO burger\_runner VALUES (2, '2021-01-03');

INSERT INTO burger\_runner VALUES (3, '2021-01-08');

INSERT INTO burger\_runner VALUES (4, '2021-01-15');

**Creating Runner Orders Table:**

CREATE TABLE runner\_orders (

order\_id INTEGER NOT NULL PRIMARY KEY,

runner\_id INTEGER NOT NULL,

pickup\_time DATETIME,

distance VARCHAR(7),

duration VARCHAR(10),

cancellation VARCHAR(23)

);

ALTER TABLE runner\_orders

ADD CONSTRAINT fk\_runner\_orders\_runner

FOREIGN KEY (runner\_id) REFERENCES burger\_runner(runner\_id);

**Inserting Values In The Table:**

INSERT INTO runner\_orders VALUES (1, 1, '2021-01-01 18:15:34', '20km', '32 minutes', NULL);

INSERT INTO runner\_orders VALUES (2, 1, '2021-01-01 19:10:54', '20km', '27 minutes', NULL);

INSERT INTO runner\_orders VALUES (3, 1, '2021-01-03 00:12:37', '13.4km', '20 mins', NULL);

INSERT INTO runner\_orders VALUES (4, 2, '2021-01-04 13:53:03', '23.4', '40', NULL);

INSERT INTO runner\_orders VALUES (5, 3, '2021-01-08 21:10:57', '10', '15', NULL);

INSERT INTO runner\_orders VALUES (6, 3, NULL, NULL, NULL, 'Restaurant Cancellation');

INSERT INTO runner\_orders VALUES (7, 2, '2021-01-08 21:30:45', '25km', '25mins', NULL);

INSERT INTO runner\_orders VALUES (8, 2, '2021-01-10 00:15:02', '23.4 km', '15 minute', NULL);

INSERT INTO runner\_orders VALUES (9, 2, NULL, NULL, NULL, 'Customer Cancellation');

INSERT INTO runner\_orders VALUES (10, 1, '2021-01-11 18:50:20', '10km', '10minutes', NULL);

**Creating Burger Names Table:**

CREATE TABLE burger\_names (

burger\_id INTEGER NOT NULL PRIMARY KEY,

burger\_name VARCHAR(10) NOT NULL

);

**Inserting Values In The Table:**

INSERT INTO burger\_names (burger\_id, burger\_name) VALUES (1, 'Meatlovers');

INSERT INTO burger\_names (burger\_id, burger\_name) VALUES (2, 'Vegetarian');

**Joins**

* **Definition**: Joins are used to combine rows from two or more tables based on a related column between them. They are crucial for querying data spread across multiple tables.
* **Types of Joins**:
  + **INNER JOIN**: Returns only the rows that have matching values in both tables.
  + **LEFT JOIN (LEFT OUTER JOIN)**: Returns all rows from the left table and the matching rows from the right table. If there's no match, NULL values are returned for columns from the right table.
  + **RIGHT JOIN (RIGHT OUTER JOIN)**: Returns all rows from the right table and the matching rows from the left table. If there's no match, NULL values are returned for columns from the left table.
  + **FULL JOIN (FULL OUTER JOIN)**: Returns all rows when there is a match in either left or right table. Non-matching rows will contain NULL values for the missing parts.
  + **CROSS JOIN**: Returns the Cartesian product of both tables (every combination of rows).
  + **SELF JOIN**: A table is joined with itself to compare rows within the same table.

**Query 1: List All Orders with Customer and Burger Names?**

SELECT co.order\_id, co.customer\_id, bn.burger\_name, co.order\_time

FROM customer\_orders co

JOIN burger\_names bn ON co.burger\_id = bn.burger\_id;

**Explanation**: This query joins the customer\_orders and burger\_names tables using the burger\_id to get the burger name associated with each order. This way, we can display each order with its corresponding customer and burger details.

A screenshot of a computer

Description automatically generated

**Query 2: Retrieve Orders with Runner Details?**

SELECT ro.order\_id, br.runner\_id, br.registration\_date, ro.pickup\_time

FROM runner\_orders ro

JOIN burger\_runner br ON ro.runner\_id = br.runner\_id;

**Explanation**: This query joins runner\_orders with burger\_runner on runner\_id to fetch the runner details like their ID, registration date, and the order's pickup time.

A screenshot of a computer

Description automatically generated

**Query 3: Fetch Orders That Were Cancelled and the Reason for Cancellation?**

SELECT ro.order\_id, ro.cancellation

FROM runner\_orders ro

WHERE ro.cancellation IS NOT NULL;

**Explanation**: A simple query filtering the runner\_orders table to list orders with a cancellation reason. The WHERE clause is used to check for non-NULL cancellation fields

A screen shot of a computer error

Description automatically generated

**Subqueries**

* **Definition**: A subquery is a query nested inside another query. Subqueries can be used to perform operations that depend on results from other queries.
* **Types of Subqueries**:
  + **Single-row Subquery**: Returns a single row (e.g., used with =, <, >, etc.).
  + **Multiple-row Subquery**: Returns multiple rows (e.g., used with IN, ANY, ALL, etc.).
  + **Correlated Subquery**: A subquery that uses values from the outer query.

**Query 4: Subquery to Find Runners Who Delivered Orders?**

SELECT runner\_id, registration\_date

FROM burger\_runner

WHERE

runner\_id IN (

SELECT DISTINCT runner\_id FROM runner\_orders WHERE cancellation IS NULL

);

**Explanation**: This query uses a subquery to get runners who delivered orders (i.e., orders without cancellation). The IN clause checks if a runner ID appears in the subquery's result.

A screenshot of a computer

Description automatically generated

**Subtotal**

* **Definition**: A subtotal is an intermediate calculation that adds up a set of values before arriving at a final total. In SQL, subtotals can be computed using arithmetic operations on columns and may include conditions to handle discounts or additional charges.
* **Usage**: Commonly used in financial calculations or reports to summarize data before applying more complex aggregations.

**Query 5: Calculate Subtotal for Each Order?**

SELECT co.order\_id, co.customer\_id, bn.burger\_name,

(100 + COALESCE(LEN(co.extras), 0) \* 10 - COALESCE(LEN(co.exclusions), 0) \* 5) AS subtotal

FROM customer\_orders co

JOIN burger\_names bn ON co.burger\_id = bn.burger\_id;

**Explanation**: This query calculates a subtotal based on arbitrary pricing logic: a base price of 100, with extras adding 10 per character and exclusions subtracting 5 per character. The COALESCE function handles NULL values to prevent errors.

A screenshot of a computer

Description automatically generated

**GROUP BY Clause**

* **Definition**: GROUP BY is used to arrange identical data into groups. This clause is often used with aggregate functions like COUNT, SUM, AVG, MAX, or MIN to perform calculations on each group.
* **Use Case**: Summarizing data to see totals or averages for specific categories.

**HAVING Clause**

* **Definition**: HAVING is used to filter groups of data produced by the GROUP BY clause based on a condition. Unlike WHERE, which filters rows before grouping, HAVING filters groups after aggregation.
* **Use Case**: Used to apply conditions to the aggregated results.

**Query 6: Count Orders Grouped by Each Burger Type?**

SELECT bn.burger\_name, COUNT(co.order\_id) AS total\_orders

FROM customer\_orders co

JOIN burger\_names bn ON co.burger\_id = bn.burger\_id

GROUP BY bn.burger\_name;

**Explanation**: This query groups orders by burger type and counts the number of orders for each burger. The GROUP BY clause is used to aggregate data per burger.

A screenshot of a computer program

Description automatically generated

**Query 7: Find Runners Who Completed More Than One Delivery?**

SELECT ro.runner\_id, COUNT(ro.order\_id) AS completed\_deliveries

FROM runner\_orders ro

WHERE ro.cancellation IS NULL

GROUP BY ro.runner\_id

HAVING COUNT(ro.order\_id) > 1;

**Explanation**: This query uses GROUP BY to group orders by runner\_id and counts only completed deliveries (those without cancellations). The HAVING clause filters for runners who have more than one completed delivery.

A screenshot of a computer

Description automatically generated