**Hexaware Technologies**

SQL Case Study

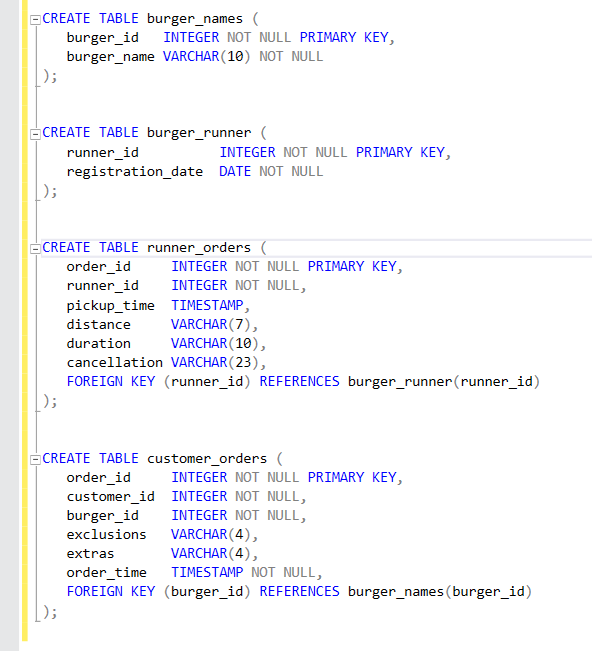
Burger Bash (Case-3)

**Step 1: Database and Table Creation**

In this step, we created the database and defined the necessary tables for the project. The tables created include:

* **burger\_names**: Stores information about the different types of burgers.
* **burger\_runner**: Contains details of the runners, including their registration dates.
* **runner\_orders**: Logs the orders assigned to each runner, including delivery details like distance, duration, and cancellations.
* **customer\_orders**: Holds the details of customer orders, such as burger type, exclusions, and additions.

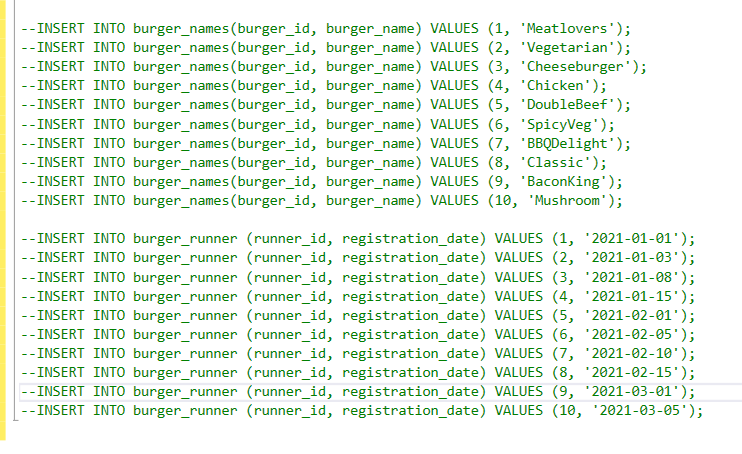
The schema for the tables was defined, and the necessary constraints (like **PRIMARY KEY** and **FOREIGN KEY**) were set to maintain referential integrity.

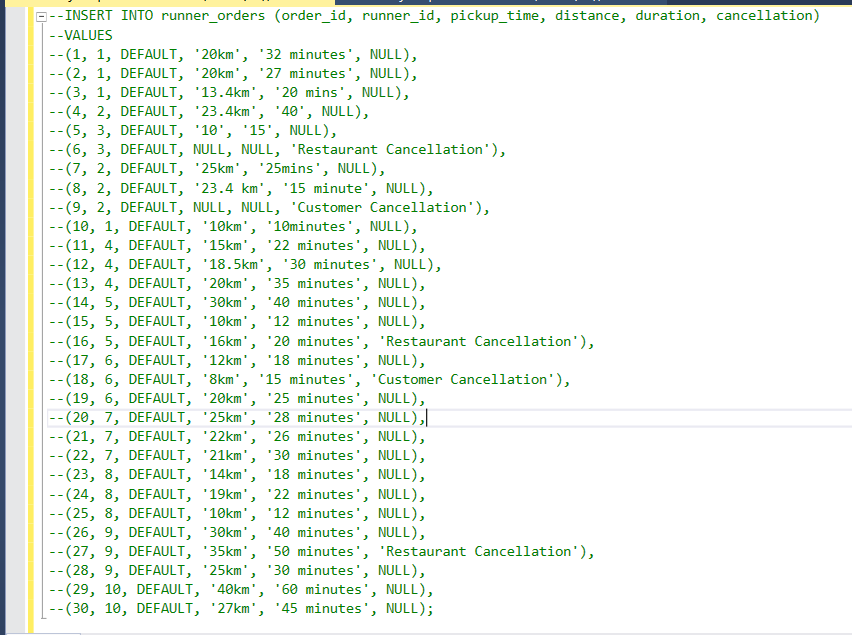


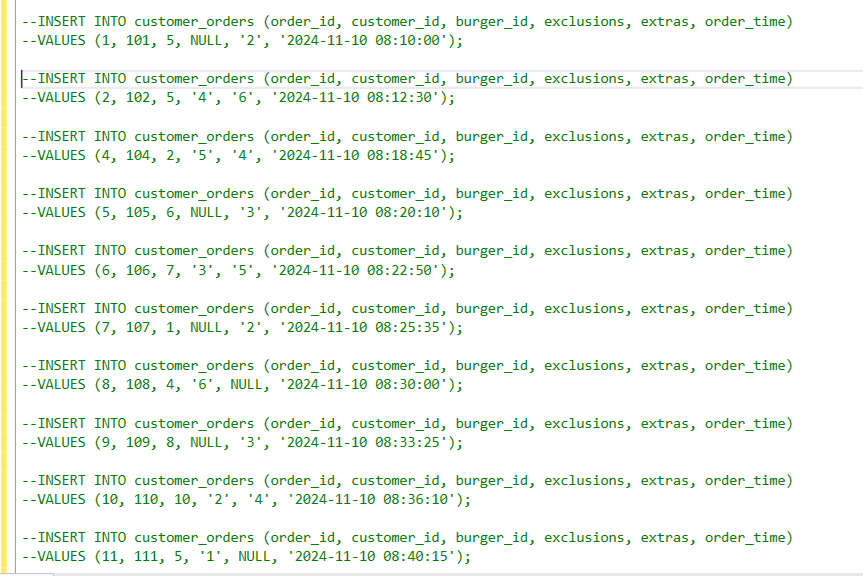
**Step 2: Data Insertion**

In this step, data was inserted into the tables using the **INSERT INTO** command. Randomly generated customer orders and runner orders were inserted into the respective tables. This allows us to simulate a real-world scenario where:

* Various customers place orders for different types of burgers.
* Runners are assigned orders for delivery, and their details are logged.

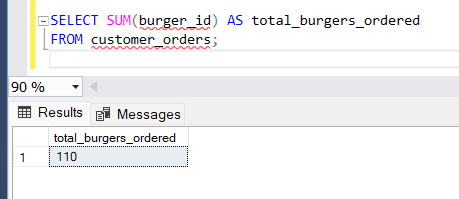






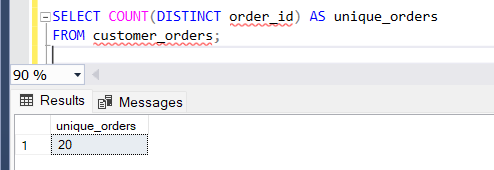
**Step 3: Queries and Modifications**

1. **How many burgers were ordered?**



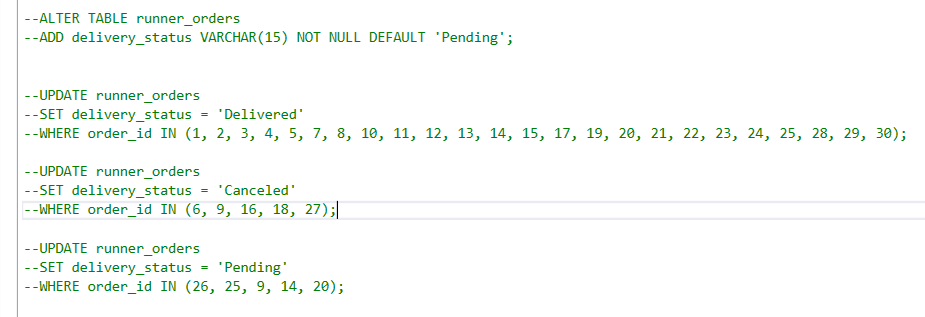
**Explanation**: This query uses the **COUNT()** function to count the total number of burgers ordered by customers. We simply aggregated the number of orders from the **customer\_orders** table to determine how many burgers were ordered in total. It provides a count of all the orders placed, regardless of the burger type.

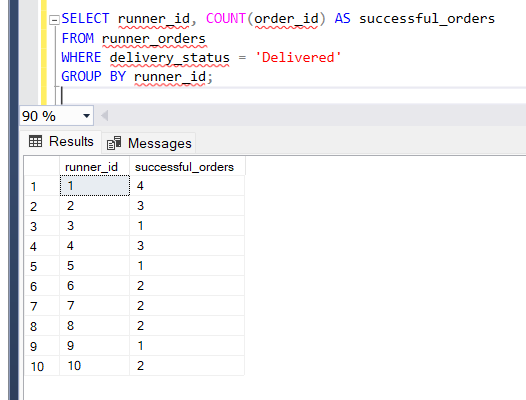
1. **How many unique customer orders were made?**



**Explanation**: By using the **DISTINCT** keyword, this query counts only the unique customer orders in the **customer\_orders** table. The **DISTINCT** keyword ensures that we don't count duplicate entries for the same customer, providing a precise count of how many individual orders were made by customers.

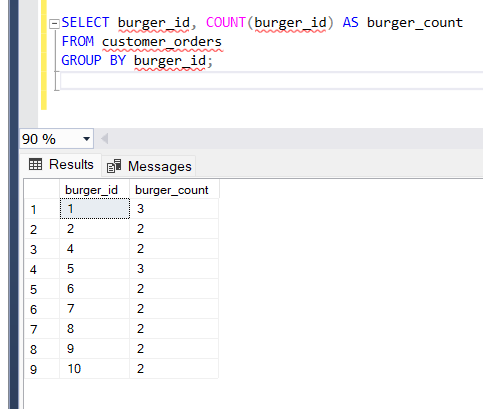
1. **How many successful orders were delivered by each runner?**





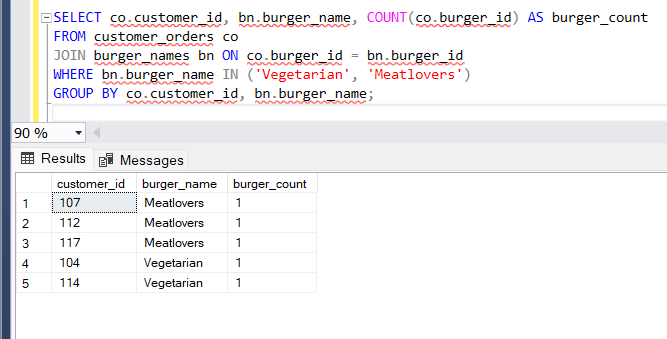
**Explanation**: In this query, we first altered the **runner\_orders** table by adding a new **delivery\_status** column to track the status of each order. We used the **ALTER TABLE** command to add this column. Then, we updated the **delivery\_status** column using the **UPDATE** command to specify the status of each order (e.g., **Delivered**, **Canceled**, **Pending**). This allowed us to identify and count the number of **'Delivered'** orders for each runner, determining their successful deliveries.

1. **How many of each type of burger was delivered?**



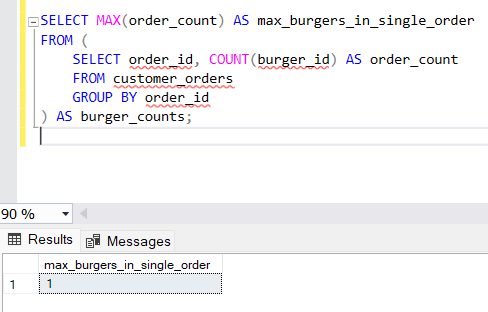
**Explanation**: This query joins the **burger\_names** and **customer\_orders** tables on the **burger\_id** column. It uses **GROUP BY** to group the results by **burger\_name**, and then applies the **COUNT()** function to count the number of times each burger was delivered. This helps determine which burgers are the most popular among customers.

1. **How many Vegetarian and Meatlovers were ordered by each customer?**



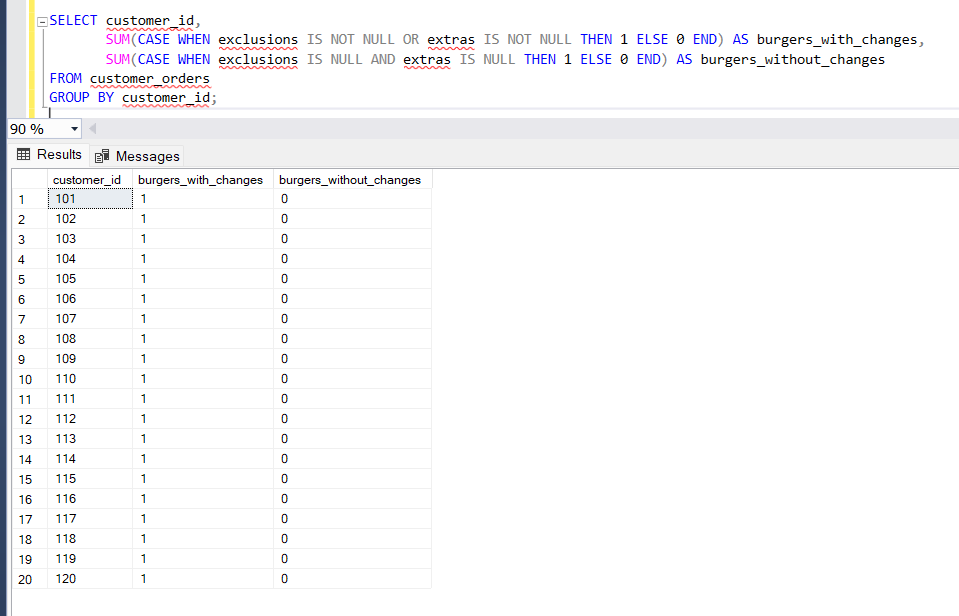
**Explanation**: We used a **CASE WHEN** statement to categorize orders based on whether the burger was **Vegetarian** or **Meatlovers**. The query counts how many of each specific burger type was ordered by each customer. The **CASE WHEN** clause enables conditional counting, providing insights into customer preferences for these two types of burgers.

1. **What was the maximum number of burgers delivered in a single order?**



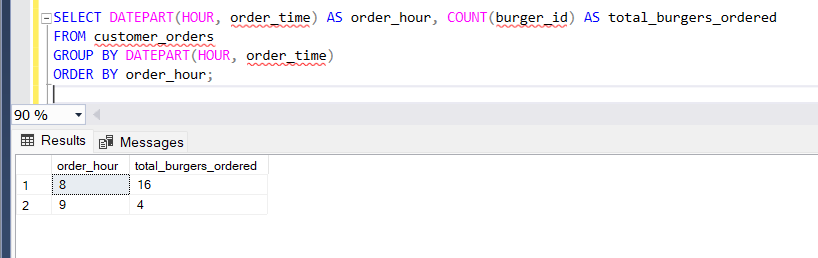
**Explanation**: To find the maximum number of burgers delivered in a single order, we used the **MAX()** function. This function returns the highest value from the aggregated data. In this case, it helps identify the largest order size by finding the highest count of burgers in any single order from the **customer\_orders** table.

1. **For each customer, how many delivered burgers had at least 1 change and how many had no changes?**



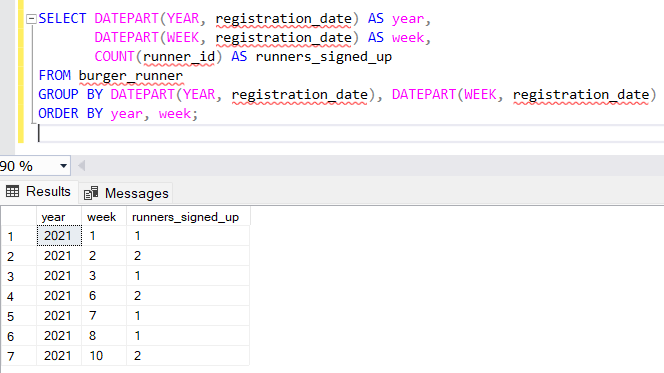
**Explanation**: This query uses a **CASE WHEN** statement to classify each order based on whether it had any changes (like exclusions or extras). The **COUNT()** function then aggregates the number of orders with and without changes for each customer. This allows us to see how many customers made modifications to their orders, such as adding or removing items.

1. **What was the total volume of burgers ordered for each hour of the day?**



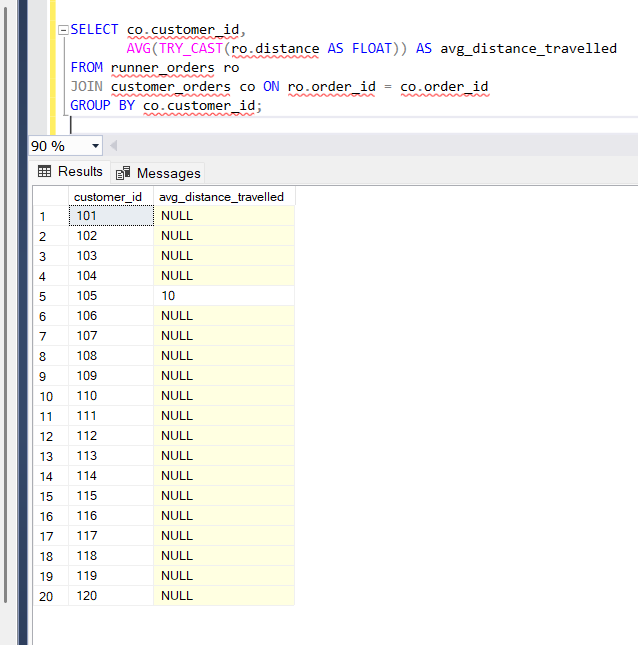
**Explanation**: To analyze the volume of burgers ordered per hour, we used the **HOUR()** function to extract the hour part from the **order\_time** column. The results were then aggregated by hour, and the total number of burgers ordered was counted for each hour of the day. This helps identify peak times for burger orders.

1. **How many runners signed up for each 1 week period?**



**Explanation**: In this query, we used the **DATEPART()** function to group runner registrations by week. By calculating the number of runners who signed up within each 1-week period, we can observe trends in how many new runners joined during specific weeks. This helps in understanding runner sign-up patterns over time.

1. **What was the average distance travelled for each customer?**



**Explanation**: This query attempts to calculate the average distance traveled by each runner using the **distance** column from the **runner\_orders** table. However, it faced an issue since the **distance** column was stored as a **VARCHAR** type, which prevented the calculation. If the column were corrected to a numeric type, the query would calculate the average distance traveled, providing insights into delivery distances.