VICTORIA UNIVERSITY
MELBOURNE AUSTRALIA

VICTORIA UNIVERSITY BUSINESS SCHOOL

BCO7004: Business Data Mining and Warehousing

<u>ASSESSMENT 2</u>

PROJECT REPORT

Student Name: Vrishank Mani (s4680492)

Code Explanation and Analysis

Q 1. List all products and their categories with sales greater than 100 units

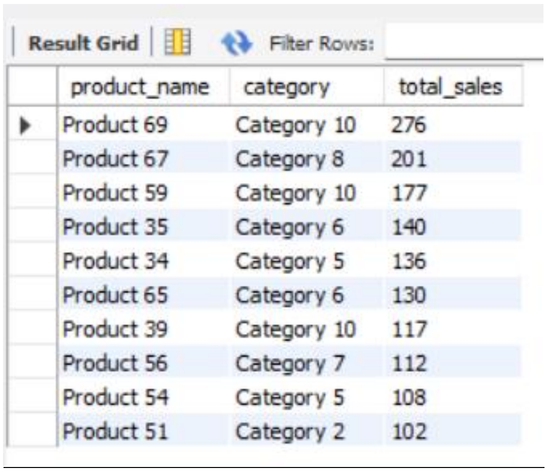
```
SELECT pd.name AS product_name, pd.category, SUM(s.product_ID) AS total_sales
FROM Sales s
JOIN ProductDetails pd ON s.product_ID = pd.product_ID
GROUP BY pd.product_ID, pd.name, pd.category
HAVING total_sales > 100
ORDER BY total_sales DESC;
```

Explanation-

To achieve this task, we need to join the 'Sales' table with the 'ProductDetails' table as these are the two tables that contain the columns needed to find all products with sales greater than 100 units.

First, using the 'SELECT' statement, I selected the 'name' and 'category' columns from the 'ProductDetails' tables. Then, using the 'SUM()' aggregate function, I selected the sum of the 'product_ID' column from the sales table to calculate the total sales. It's important to mention here that I used the 'AS' keywork to create an alias 'total_sales' for the sum of the product_IDs column. I then joined the two tables using the JOIN clause on the 'Product_ID' column, which is the PK for the ProductDetails column and the FK for the 'Sales' column. I then grouped the columns by the 'product_ID', 'name' and 'category' columns using the GROUP BY clause. Finally using the HAVING clause, I filtered the rows for products with total sales greater than 100 units. The reason I used the 'HAVING()' clause instead of the 'WHERE()' clause is because the WHERE() clause is applied to individual rows which are not grouped. Here however, the 'total_sales' column is grouped by the 'product_ID', 'product_name' and 'category' columns and so 'HAVING()' is the appropriate clause to use.

Result:



product_name	category	total_sales
Product 69	Category 10	276
Product 67	Category 8	201
Product 59	Category 10	177
Product 35	Category 6	140
Product 34	Category 5	136
Product 65	Category 6	130
Product 39	Category 10	117
Product 56	Category 7	112
Product 54	Category 5	108
Product 51	Category 2	102

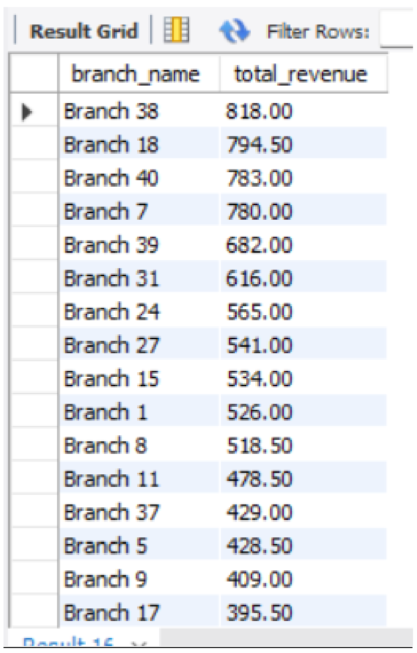
Based on the output, we can clearly see there are 10 products with total sales > 100, and Product 69 has the highest sales with 276 units sold. By looking at the category column, we can also see that there are 3 products from category 10 and 2 products from category 5 and category 6. This acts as an indicator for AussieRetailers that they should stock more items from these categories as they lead to the most sales.

Q 2. Calculate the total revenue per branch, considering the quantity sold and product prices

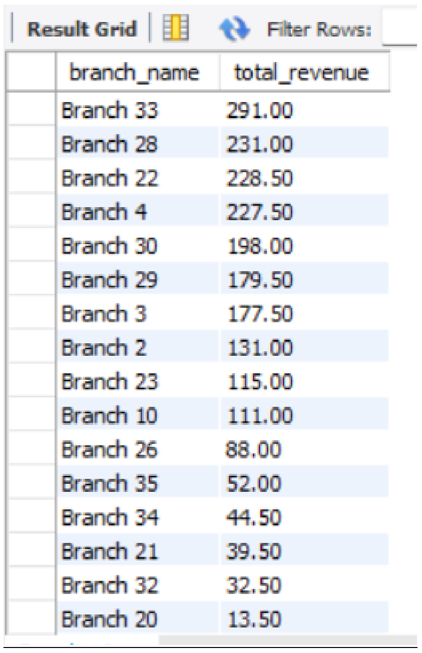
```
SELECT b.branch_name, SUM(s.quantity * p.price) AS total_revenue
FROM Sales s
JOIN ProductDetails p ON s.product_ID = p.product_ID
JOIN BranchDetails b ON s.branch_ID = b.branch_ID
GROUP BY b.branch_ID, b.branch_name
ORDER BY SUM(s.quantity * p.price) DESC;
```

Explanation-
To calculate revenue, we need to join the ‘Sales’ table with the ‘ProductDetails’ table to obtain the price of each product. Then we need to multiply that by the quantity sold of each product and group it by branch.
I first selected the ‘name’ from the branch table. Then, using the aggregate function, ‘SUM()’, I calculated the sum of the product of the quantity and price columns to get the ‘total_revenue’ (alias name). To do this, I first joined the ‘ProductDetails’ table with the ‘Sales’ table on the ‘product_ID’ column. Then, I joined the ‘BranchDetails’ table with the ‘Sales’ table on the ‘branch_ID’ column. Finally, I grouped the columns by the ‘branch_ID’ and ‘branch_name’ columns using the GROUP BY clause and sorted the results by the total revenue in descending order using the ORDER BY clause.

Result:



branch_name	total_revenue
Branch 38	818.00
Branch 18	794.50
Branch 40	783.00
Branch 7	780.00
Branch 39	682.00
Branch 31	616.00
Branch 24	565.00
Branch 27	541.00
Branch 15	534.00
Branch 1	526.00
Branch 8	518.50
Branch 11	478.50
Branch 37	429.00
Branch 5	428.50
Branch 9	409.00
Branch 17	395.50



branch_name	total_revenue
Branch 33	291.00
Branch 28	231.00
Branch 22	228.50
Branch 4	227.50
Branch 30	198.00
Branch 29	179.50
Branch 3	177.50
Branch 2	131.00
Branch 23	115.00
Branch 10	111.00
Branch 26	88.00
Branch 35	52.00
Branch 34	44.50
Branch 21	39.50
Branch 32	32.50
Branch 20	13.50

This is just a snippet of the top and bottom rows of the result table. We can clearly see that Branch 38 brings in the most revenue at 818.00. Meanwhile Branch 20 was the lowest performing branch in 2022 as it brought in a total revenue of just 13.50. This is a sign for AussieRetailers to either invest and focus more on the lower performing branches to improve their total revenue or, shut them down completely to save on the operating cost.

Q 3. Identify customers who have made purchases in more than 3 different branches.

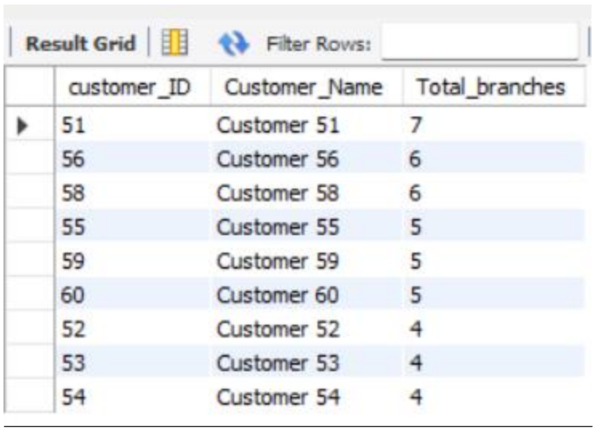
```
SELECT s.customer_ID, c.name AS Customer_Name, count(DISTINCT s.branch_ID) AS
Total_branches
FROM Sales s
JOIN CustomerDetails c ON c.customer_ID = s.customer_ID
GROUP BY s.customer_ID
HAVING count(DISTINCT s.branch_ID) > 3
ORDER BY Total_branches DESC;
```

Explanation-

In order to do this, we need to count the distinct branches at which each customer made their purchase in and then filter for those with more than 3 distinct branches.

I first selected the 'customer_ID' and 'name' columns from 'Sales' and 'CustomerDetails' tables respectively. Then using the 'COUNT()' aggregate function and the DISTINCT statement, I selected the count of all unique 'branch_ID' from the 'Sales' table. I then joined the two tables using the 'JOIN()' clause on the 'customer_ID' column. Next, I grouped the results by the 'customer_ID' column and filtered them using the 'HAVING()' clause to obtain the required results. I finally sorted the results using the 'ORDER BY()' clause to get the results in descending order of most branches shopped at.

Result:



customer_ID	Customer_Name	Total_branches
51	Customer 51	7
56	Customer 56	6
58	Customer 58	6
55	Customer 55	5
59	Customer 59	5
60	Customer 60	5
52	Customer 52	4
53	Customer 53	4
54	Customer 54	4

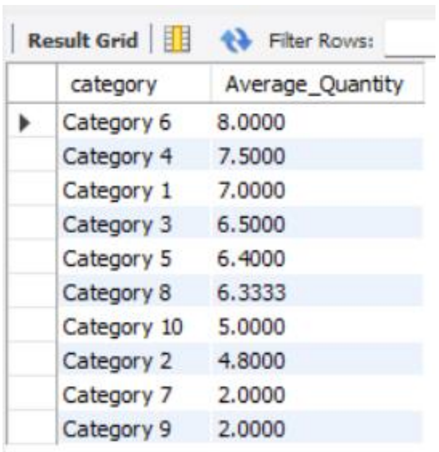
The results show that there are 9 different customers who shop at more than 3 branches. Off those, Customer 51 is a loyal customer who has shopped at 7 different AussieRetailers branches. AussieRetailers can go deeper into its analysis and find the reason why some customers shop at so many different branches. If it is proximity to certain residential zones, then AussieRetailers can open more branches in those zones where there is demand, but no stores.

Q 4. Determine the average sale quantity of products by category for sales made in the last quarter

```
SELECT pd.category, AVG(s.quantity) AS Average_Quantity
FROM Sales s
JOIN ProductDetails pd ON s.product_ID = pd.product_ID
WHERE QUARTER(s.sale_date) = 1
GROUP BY pd.category
ORDER BY Average_Quantity DESC;
```

Explanation-
We need to identify all sales made in the last quarter, and join the 'Sales' and 'ProductDetails' tables to get the category of each product and then calculate the average sale for each category. I first selected the 'category' column from the 'ProductDetails' table. Then, using the 'AVG()' aggregate function, I selected the average of the 'quantity' column from the 'Sales' table. I then joined the two tables on the 'product_ID' column and using the 'WHERE' clause, I filtered the rows to obtain all sales in the last quarter. In order to do this, I used the 'QUARTER()' function on the 'sale_date' column and set it equal to 1. I finally grouped the results by the 'category' column and sorted them in descending order of average quantity to achieve the desired result.

Result:



category	Average_Quantity
Category 6	8.0000
Category 4	7.5000
Category 1	7.0000
Category 3	6.5000
Category 5	6.4000
Category 8	6.3333
Category 10	5.0000
Category 2	4.8000
Category 7	2.0000
Category 9	2.0000

The result table shows that Category 6 has the highest average quantity sold at 8, while categories 9 and 7 have an average quantity sold of only 2.

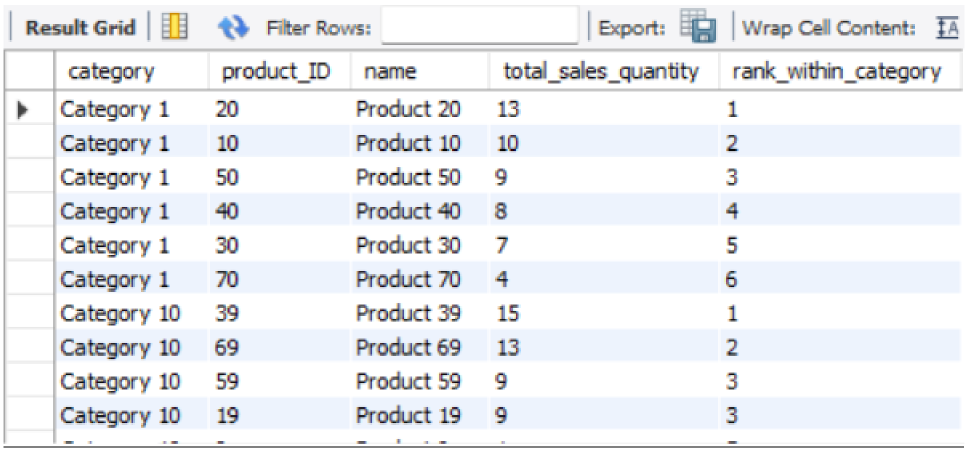
Q 5. Rank products within each category based on the total sales quantity using a window function.

```
SELECT pd.category, pd.product_ID, pd.name, SUM(s.quantity) AS total_sales_quantity,
        RANK() OVER (PARTITION BY pd.category ORDER BY SUM(s.quantity) DESC) AS
        rank_within_category
FROM ProductDetails pd
JOIN Sales s ON pd.product_ID = s.product_ID
GROUP BY pd.category, pd.product_ID, pd.name
ORDER BY pd.category, rank_within_category;
```

Explanation:

This query uses a window function, which performs a calculation across a set of table rows that are somehow related to the current row. But unlike aggregate functions, window functions don't cause rows to become grouped into a single output row. In this case, we use the 'RANK()' function to assign ranks to products based on their total sales quantity within each category. First, we select the 'category', 'product_ID', 'product_name' and the 'total_sales_quantity' (alias name) columns. Then using the 'RANK() OVER (PARTITION BY pd.category ORDER BY SUM(s.quantity) DESC)' function we rank the products within each category based on the total sales quantity. Here:
'PARTITION BY pd.category' divides the result set into partitions (one for each category).
'ORDER BY SUM(s.quantity) DESC' Orders the rows within each partition by the total sales quantity in descending order.
And finally, 'RANK()' assigns a rank to each row within the partition.
Once this is done, we join the 'ProductDetails' table with the 'Sales' table using the 'JOIN()' clause on the 'product_ID' column. Finally, we group the results by the 'category', 'product_ID' and 'name' columns and order the results by category and rank.

<u>Results:</u>



category	product_ID	name	total_sales_quantity	rank_within_category
Category 1	20	Product 20	13	1
Category 1	10	Product 10	10	2
Category 1	50	Product 50	9	3
Category 1	40	Product 40	8	4
Category 1	30	Product 30	7	5
Category 1	70	Product 70	4	6
Category 10	39	Product 39	15	1
Category 10	69	Product 69	13	2
Category 10	59	Product 59	9	3
Category 10	19	Product 19	9	3

This is just a snippet of the results table. We can clearly see that within Category 1, product 20 has the highest sales quantity at 13 units while product 70 has the lowest sales quantity within Category 1 at 4 units sold. AussieRetailers can use this table of information to understand which products within each category are selling well and which products are not. This can help with stocking decisions and also in coming up with sales and marketing strategies for products that aren't doing well.

Q 6. Show the month-over-month percentage growth in sales for the top 5 products by total quantity sold

```
WITH TopProducts AS (
        SELECT s.product_ID, p.name AS product_name, SUM(s.quantity) AS
        total_quantity
        FROM Sales s
        JOIN ProductDEtails p ON s.product_ID = p.product_ID
        GROUP BY s.product_ID, p.name
        ORDER BY total_quantity DESC
        LIMIT 5
        ),
MonthlySale AS (
        SELECT p.product_ID, p.product_name, sum(s.quantity) AS monthly_quantity,
        DATE_FORMAT(s.sale_date, '%Y-%m') AS sale_month
        FROM Sales s
        JOIN TopProducts p ON p.product_ID = s.product_ID
        GROUP BY p.product_ID, p.product_name, sale_month
        ),
    MonthlyGrowth AS (
        SELECT ms.product_ID, ms.product_name, ms.sale_month,
        ms.monthly_quantity,
        LAG(ms.monthly_quantity) OVER (PARTITION BY ms.product_ID ORDER BY
        ms.sale_month) AS previous_month_quantity,
        ((ms.monthly_quantity - LAG(ms.monthly_quantity) OVER (PARTITION BY
        ms.product_ID ORDER BY ms.sale_month)) / LAG(ms.monthly_quantity) OVER
        (PARTITION BY ms.product_ID ORDER BY ms.sale_month)) * 100 AS
        percentage_growth
        FROM  MonthlySale ms
        )
    SELECT product_ID, product_name, sale_month, monthly_quantity,
    previous_month_quantity, percentage_growth
    FROM MonthlyGrowth
    WHERE previous_month_quantity IS NOT NULL
    ORDER BY product_ID, sale_month;
```

Explanation-
In order to solve this question, we need to use Common Table Expressions (CTE). CTEs are used when we use the same subquery multiple times within our query. It helps with referencing the derived table again and again in a single query. In this query, we create 3 CTEs: TopProducts, MonthlySales, and MonthlyGrowth.

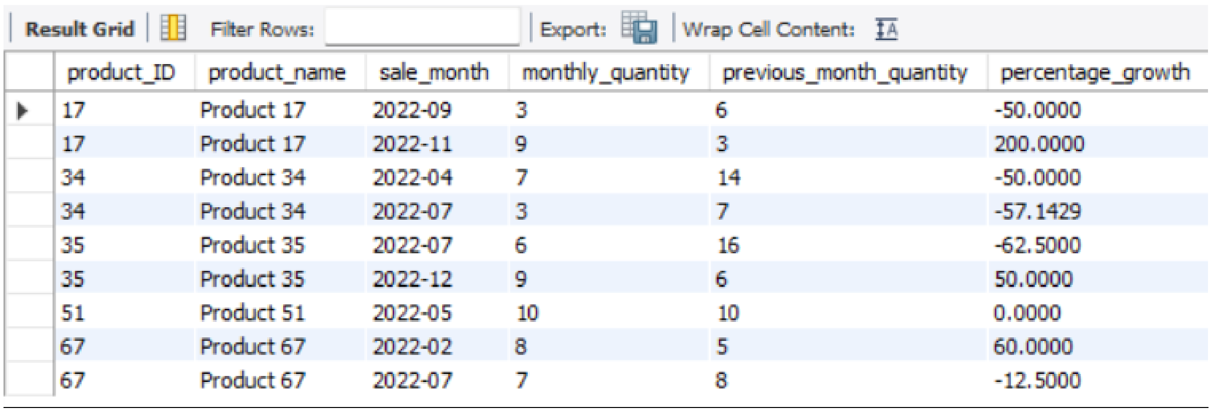
TopProducts CTE Subquery: This first subquery identifies the top 5 products by total quantity sold. The 'WITH' statement is used to define the CTE named 'TopProducts'. We then select the required columns, including the 'SUM()' aggregate of the 'quantity' column to obtain the 'total_quantity' of items. We then join the appropriate tables and group by the 'product_ID' and 'product_name' columns. Finally, we order our results in descending order of total quantity and use the 'LIMIT' clause to obtain just the top 5 results.

MonthlySales CTE Subquery: This subquery calculates the monthly sales for the top 5 products. Once we select the required columns, and use the 'DATE_FORMAT()' function to set the date to the 'YY-MM' format, we join the 'Sales' column with previously defined 'TopProduct' CTE to filter sales data for only the top 5 products. Finally, we group by the 'product_ID', 'product_name' and 'sale_month' columns to calculate the monthly quantities sold.

MonthlyGrowth CTE Subquery: This subquery calculate the month-over month percentage growth. This subquery uses a window function. In this case, we use the 'LAG' window function to get the monthly quantity of the previous month for each product, portioned by (using PARTITION BY) 'product_ID' and ordered by (using ORDER BY) the sale month. We then calculate the month-over-month percentage growth using the current month's quantity and the previous month's quantity.

We finally select the relevant columns from the 'MonthlyGrowth' CTE and use the WHERE clause to filter rows where the 'previous_month_quantity' is 'NOT NULL'

Result:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

product_ID	product_name	sale_month	monthly_quantity	previous_month_quantity	percentage_growth
17	Product 17	2022-09	3	6	-50.0000
17	Product 17	2022-11	9	3	200.0000
34	Product 34	2022-04	7	14	-50.0000
34	Product 34	2022-07	3	7	-57.1429
35	Product 35	2022-07	6	16	-62.5000
35	Product 35	2022-12	9	6	50.0000
51	Product 51	2022-05	10	10	0.0000
67	Product 67	2022-02	8	5	60.0000
67	Product 67	2022-07	7	8	-12.5000

From the results, we can clearly see that Product 17 had the highest percentage month-over-month growth of 200%, while Product 34 had the lowest percentage month-over-month growth of -57.1%.

Q 7. Find all products that have not been sold in the last 6 months (from June to December 2022) but have stock levels above 50

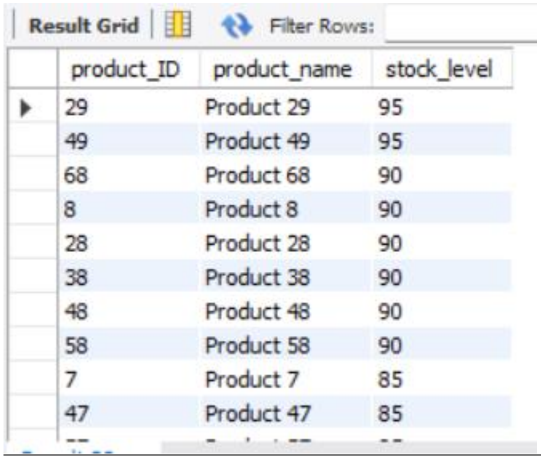
```
SELECT p.product_ID, p.name AS product_name, p.stock_level
FROM ProductDetails p
LEFT JOIN (
        SELECT s.product_ID
        FROM Sales s
        WHERE s.sale_date BETWEEN '2022-06-01' AND '2022-12-31'
        GROUP BY s.product_ID
) AS recent_sales ON p.product_ID = recent_sales.product_ID
WHERE recent_sales.product_ID IS NULL
AND p.stock_level > 50
ORDER BY p.stock_level DESC;
```

Explanation-

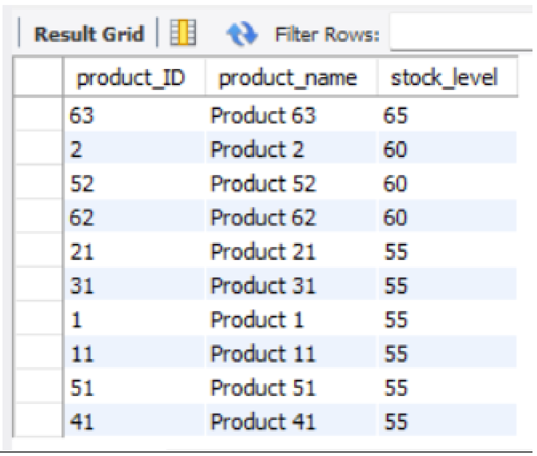
To solve this, we need to write a subquery that filters sales sales within the last 6 months, and then join it with the 'ProductDetails' table. For this question, we use the 'LEFT JOIN()' because it helps identify products with no sale during the 6 month period.

I first wrote the subquery which filters sales made between June 1, 2022 and December 31, 2022, and groups them by 'product_ID'. For this, I use the WHERE clause along with the BETWEEN statement to filter out the sales. I then use the GROUP BY clause to group the rows on the 'product_ID' column. Then, I left join the 'ProductDetails' table with the subquery. This ensures that all products from the 'ProductDetails' table are included, even if they have no corresponding sales in the subquery. Then, using the WHERE recent_sales.product_ID IS NULL statement, I filter out the products that have no sale in the last 6 months. Finally, I also use the AND function to add to the WHERE clause in which I filter the products with stock level greater than 50. This gives us the desired results.

Result:



	product_ID	product_name	stock_level
▶	29	Product 29	95
	49	Product 49	95
	68	Product 68	90
	8	Product 8	90
	28	Product 28	90
	38	Product 38	90
	48	Product 48	90
	58	Product 58	90
	7	Product 7	85
	47	Product 47	85



	product_ID	product_name	stock_level
	63	Product 63	65
	2	Product 2	60
	52	Product 52	60
	62	Product 62	60
	21	Product 21	55
	31	Product 31	55
	1	Product 1	55
	11	Product 11	55
	51	Product 51	55
	41	Product 41	55

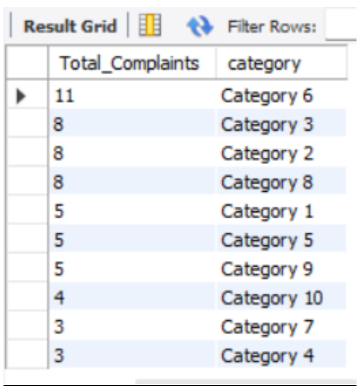
This is just a snippet of the top and bottom rows of the result table. We can clearly see that off all the products with a stock level greater than 50 not sold over the last 6 months, Products 29 and 49 have the highest stock level of 95. On the other hand, there are 5 products which have the lowest stock level over 50, at 55. This information gives AussieRetailers an idea about which products to stock less of due to lack of sales, or which products to market more in order to increase their sales. By stocking less of certain items, AussieRetailers can save on storage and operational costs. On the other hand, they may choose to advertise their low selling and high stocked items.

Q 8. Calculate the total number of complaints lodged against products in each category.

```
SELECT count(c.complaint_ID) AS Total_Complaints, pd.category
FROM ComplaintDetails c
JOIN ProductDetails pd ON pd.product_ID = c.product_ID
GROUP BY pd.category
ORDER BY count(c.complaint_ID) DESC;
```

Explanation-
Using the COUNT() aggregate function, I first selected the count of complaints along with the category. I then joined the two tables involved, 'ComplaintDetails' and 'ProductDetails' and grouped them by category. Finally I sorted them in descending order of complaint counts to get the desired result.

Result:



	Total_Complaints	category
▶	11	Category 6
	8	Category 3
	8	Category 2
	8	Category 8
	5	Category 1
	5	Category 5
	5	Category 9
	4	Category 10
	3	Category 7
	3	Category 4

The result shows that Category 6 generated the most complaints at 11, while categories 7 and 4 generated the least complaints at 3. This gives AussieRetailers an insight into which categories of products customers are the least happy with. Hence, they can invest resources focused on improving their products in those categories.

Q 9. List the top 10 customers by total spending and show their most frequently bought product category

```
WITH TotalSpending AS (
        SELECT s.customer_ID, SUM(p.price * s.quantity) AS total_spending
        FROM Sales s
        JOIN ProductDetails p ON s.product_ID = p.product_ID
        GROUP BY s.customer_ID
),
TopCustomers AS (
        SELECT customer_ID, total_spending
        FROM TotalSpending
        ORDER BY total_spending DESC
        LIMIT 10
),
CustomerCategories AS (
        SELECT s.customer_ID, p.category, COUNT(*) AS purchase_count,
        ROW NUMBER() OVER (PARTITION BY s.customer_ID ORDER BY COUNT(*)
        DESC) AS rn
        FROM Sales s
        JOIN ProductDetails p ON s.product_ID = p.product_ID
        GROUP BY s.customer_ID, p.category
)
SELECT tc.customer_ID, tc.total_spending, cc.category AS most_frequent_category,
cc.purchase_count
FROM TopCustomers tc
JOIN CustomerCategories cc ON tc.customer_ID = cc.customer_ID
WHERE cc.rn = 1
ORDER BY tc.total_spending DESC;
```

Explanation-
This question again involves the usage of CTEs. There are 3 CTEs: TotalSpending, TopCustomers and CustomerCategories

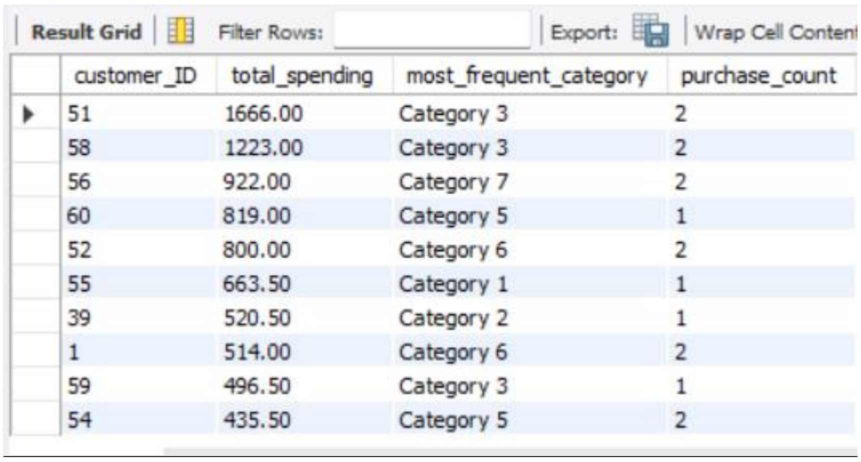
TotalSpending CTE Subquery: This CTE calculates the total spending for each customer. We first select the 'customer_ID' column along with the 'SUM()' aggregate of 'price * quantity' in order to get the total spending. We then join the 'Sales' and 'ProductDetails' tables using the JOIN clause on the 'product_ID' column. Finally, we group by the 'customer_ID' column to get the total spending per customer.

TopCustomers CTE Subquery: This subquery identifies the the top 10 customers by total spending. We select the relevant coluns from the TotalSpending CTE defined above and order the rows in descending order of the total spending amount. Then, using the 'LIMIT' clause, we select the top 10 customers.

CustomerCategories CTE subquery: This determines the most frequently bought product category for each top customer. It first calculates the purchase count of each product category for each customer, and then assigns a row number ('ROW NUMBER()') to each category per customer. This is then ordered by the purchase count in descending order using the window function ('ROW NUMBER() OVER (PARTITION BY s.customer_ID ORDER BY COUNT(*) DESC) AS rn'). Here 'rn' is used as an alias for the row number.

The final query joins the 'TopCustomers' subquery with their most frequently bought category by filtering 'CustomerCategories' to keep only the rows where the row number ('rn') is 1. The results are then ordered by total spending in descending order to get the desired result.

Result:


Result Grid | Filter Rows: | Export: | Wrap Cell Content

customer_ID	total_spending	most_frequent_category	purchase_count
51	1666.00	Category 3	2
58	1223.00	Category 3	2
56	922.00	Category 7	2
60	819.00	Category 5	1
52	800.00	Category 6	2
55	663.50	Category 1	1
39	520.50	Category 2	1
1	514.00	Category 6	2
59	496.50	Category 3	1
54	435.50	Category 5	2

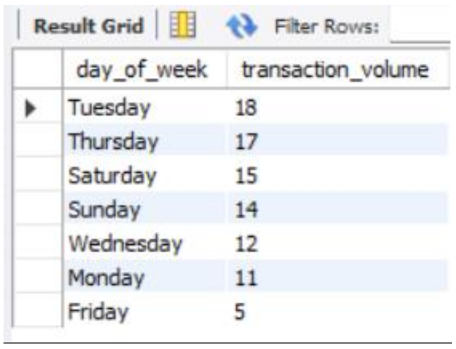
We can clearly see that customer 51 has spent the most at $1666.0 and his most frequently purchased category is Category 3 (with a frequency of 2). It is important to not that 3 customers in this list have purchased the most from category 3. This is an indication that Category 3 is popular among customers who spend a lot.

Q 10. Identify days of the week with the highest sales transactions volume.

```
SELECT DAYNAME(s.sale_date) AS day_of_week, COUNT(*) AS transaction_volume
FROM Sales s
GROUP BY day_of_week
ORDER BY transaction_volume DESC;
```

Explanation-
To solve this question, we first extract the day of the week from the 'sale_date' column in the 'Sales' table using the DAYNAME() function. We then count the number of sales transaction for each day of the week using the COUNT() function. Next, we group the rows by the day of the week and finally order them in descending order of transaction volume.

Result:



	day_of_week	transaction_volume
▶	Tuesday	18
	Thursday	17
	Saturday	15
	Sunday	14
	Wednesday	12
	Monday	11
	Friday	5

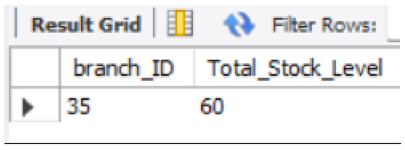
This helps AussieRetailers get an idea of which days of the week customers are shopping more and which days of the week the sales traffic is low. This can not only help with estimating the stock keeping units of products for each day of the week, but also allows in rostering employees at different branches on different days of the week based on the footfall.

Q 11. Determine the branch with the lowest stock levels across all products.

```
SELECT s.branch_ID, sum(pd.stock_level) AS Total_Stock_Level
FROM Sales s
JOIN ProductDetails pd ON s.product_ID = pd.product_ID
GROUP BY s.branch_ID
ORDER BY Total_Stock_Level ASC
LIMIT 1;
```

Explanation-
To achieve this, we first aggregate the stock levels for each branch using the SUM() function. Then we join the 'ProductDetails' table with the 'Sales' table and then group by the 'branch_ID' column. We then filter out the branch with the lowest stock by first sorting the rows in ascending order of the total stock level, and then using the LIMIT function to get the one with the lowest stock level.

Result:



	branch_ID	Total_Stock_Level
▶	35	60

We can clearly see that branch 35 has the lowest stock level of 60. Based on the footfall and total sales and revenue generated by that store, AussieRetailers can either decide to stock that branch up with more items, or they may even decide to shut that store down if it doesn't generate profits.

Q 12. Analyze the correlation between loyalty program status and the average transaction value per customer

```
WITH AvgTransactionValue AS (
        SELECT cd.customer_ID, cd.loyalty_program_status, AVG(pd.price * s.quantity)
        AS Avg_Transaction_Value
        FROM customerdetails cd
        JOIN Sales s ON cd.customer_id = s.customer_ID
        JOIN productdetails pd ON s.product_ID = pd.product_ID
        GROUP BY cd.customer_id
),
Stats AS (
        SELECT COUNT(*) AS n,  SUM(Avg_Transaction_Value) AS sum_x,
        SUM(loyalty_program_status) AS sum_y,
        SUM(Avg_Transaction_Value * loyalty_program_status) AS sum_xy,
        SUM(Avg_Transaction_Value * Avg_Transaction_Value) AS sum_xx,
        SUM(loyalty_program_status * loyalty_program_status) AS sum_yy
        FROM AvgTransactionValue
),
Correlation AS (
        SELECT (n * sum_xy - sum_x * sum_y) /
        SQRT((n * sum_xx - sum_x * sum_x) * (n * sum_yy - sum_y * sum_y)) AS
        correlation
        FROM Stats
)
SELECT correlation
FROM Correlation;
```

Explanation-

In order to analyze the correlation between loyalty program status and the average transaction value per customer, we need to use the following formula:

$$r = \frac{n\sum(xy) - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

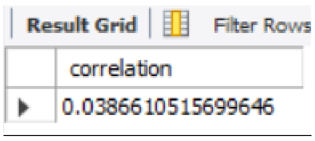
This query involves the following CTEs: AvgTransactionsValue, Stats, and Correlation.

AvgTransactionValue CTE: This calculates the average transaction value for each customer. We first select the 'customer_ID' and 'loyalty_program_status' from the 'CustomerDetails' table. We then perform 'AVG()' aggregation on the product of the 'price' and 'quantity' columns to get the 'avg_transaction_value' (alias name). We then join the 'Sales' and 'CustomerDetails' tables and the 'ProductDetails' and 'Sales' tables and group by the 'customer_ID' column.

Stats CTE: This computes the total count of records ('n'), sum of 'avg_transaction_value' ('sum_x'), sum of 'loyalty_program_status' ('sum_y'), sum of the product of 'avg_transaction_value' and 'loyalty_program_status' ('sum_xy'), sum of squares of 'avg_transaction_value' ('sum_xx'), and sum of squares of 'loyalty_program_status' ('sum_yy'). All this is done by selecting the relevant columns from the 'CustomerAvgTransaction' CTE defined above.

Correlation CTE: We use the sums and counts from the 'Stats' CTE to calculate the correlation coefficient by inputting them in the above formula.

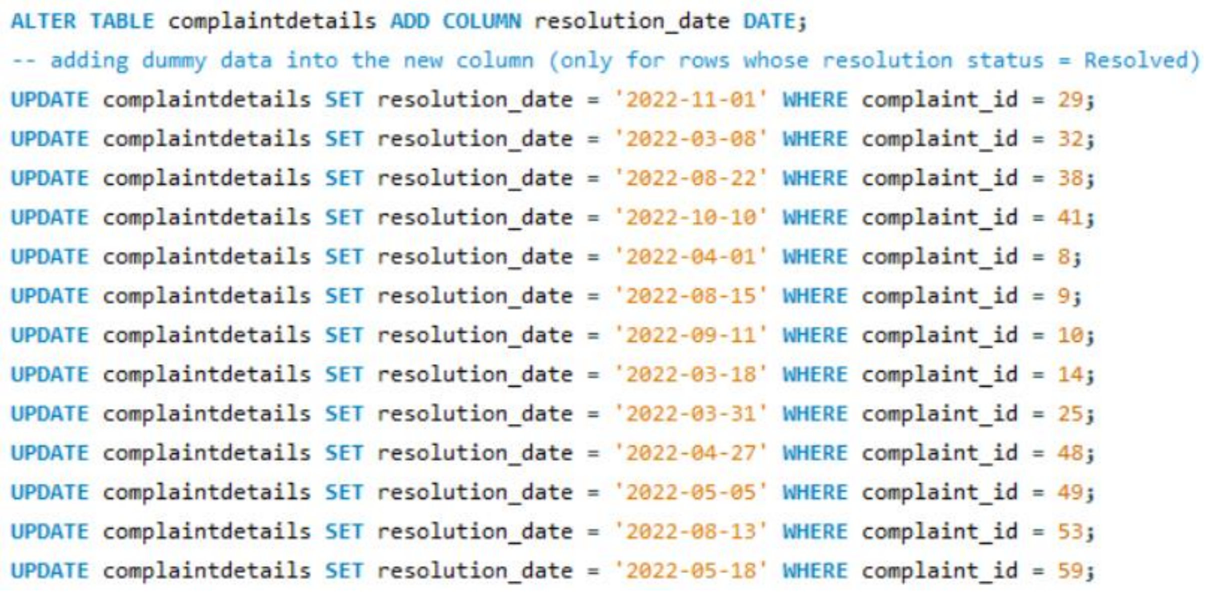
Result:



```
correlation
0.0386610515699646
```

A correlation coefficient (*r*) of 0.0386 is very close to 0. As a result, it indicates that there is a very weak linear relationship between the loyalty program status and the average transaction value per customer. Although the relationship is weak, it is slightly positive, suggesting that there is a very small tendency for customers who are part of the loyalty program to have higher transaction values compared to those who are not. Given the weak correlation, we can predict that there are likely more influential factors that determine the average transaction value of customers.

Q 13. Calculate the average duration between complaint registration and resolution.
The dataset provided for the question did not have any column for the Resolution date of complaints. Hence to solve this question, I created a new column in the 'ComplaintDetails' table and input dummy values for the resolution date. I only did this for complaints whose resolution status was marked 'Resolved'. The rest of the rows were left empty. To do this, I first added a new column called 'Resolution_date' using the 'ALTER TABLE' and 'ADD COLUMN' commands. Later I added dummy values using the 'UPDATE' and 'SET' commands.



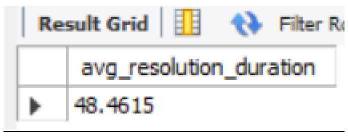
```sql
ALTER TABLE complaintdetails ADD COLUMN resolution_date DATE;
-- adding dummy data into the new column (only for rows whose resolution status = Resolved)
UPDATE complaintdetails SET resolution_date = '2022-11-01' WHERE complaint_id = 29;
UPDATE complaintdetails SET resolution_date = '2022-03-08' WHERE complaint_id = 32;
UPDATE complaintdetails SET resolution_date = '2022-08-22' WHERE complaint_id = 38;
UPDATE complaintdetails SET resolution_date = '2022-10-10' WHERE complaint_id = 41;
UPDATE complaintdetails SET resolution_date = '2022-04-01' WHERE complaint_id = 8;
UPDATE complaintdetails SET resolution_date = '2022-08-15' WHERE complaint_id = 9;
UPDATE complaintdetails SET resolution_date = '2022-09-11' WHERE complaint_id = 10;
UPDATE complaintdetails SET resolution_date = '2022-03-18' WHERE complaint_id = 14;
UPDATE complaintdetails SET resolution_date = '2022-03-31' WHERE complaint_id = 25;
UPDATE complaintdetails SET resolution_date = '2022-04-27' WHERE complaint_id = 48;
UPDATE complaintdetails SET resolution_date = '2022-05-05' WHERE complaint_id = 49;
UPDATE complaintdetails SET resolution_date = '2022-08-13' WHERE complaint_id = 53;
UPDATE complaintdetails SET resolution_date = '2022-05-18' WHERE complaint_id = 59;
```

Code for the solution:

```sql
SELECT AVG(DATEDIFF(resolution_date, complaint_date)) AS avg_resolution_duration
FROM ComplaintDetails
WHERE resolution_date IS NOT NULL;
```

Explanation-
To find the resolution duration, I first calculated the difference between the 'resolution_date' and the 'complaint_date' by using the 'DATEDIFF()' function. Then I aggregated this using the 'AVG()' function to calculate the 'avg_resolution_duration' (alias name). Finally, I used the 'WHERE' clause to filter the rows where the resolution date was not null because only the complaints which had been 'Resolved' had a resolution date.

Results:



avg_resolution_duration
48.4615

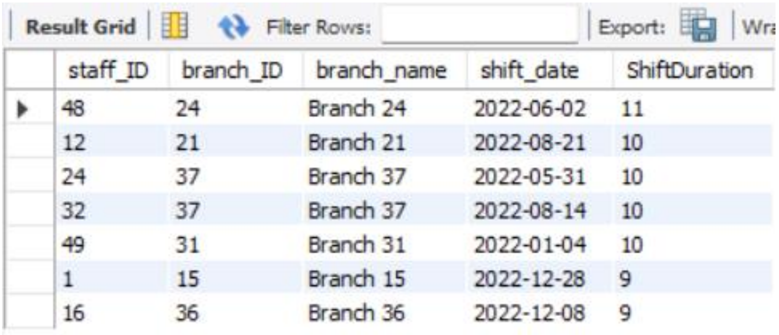
The average resolution duration was 48.46 days, which is a long period of time for retail complaints. AussieRetailers must focus on bringing the resolution duration down otherwise it may lead to customer dissatisfaction and as a result lower sales.

Q 14. Identify staff members with shifts longer than 8 hours and list their corresponding branches and shift dates.

```
SELECT ssd.staff_ID, ssd.branch_ID, b.branch_name, ssd.shift_date,
HOUR(TIMEDIFF(ssd.end_time, ssd.start_time)) AS ShiftDuration
FROM StaffShiftDetails ssd
JOIN BranchDetails b ON b.branch_ID = ssd.branch_ID
WHERE HOUR(TIMEDIFF(ssd.end_time, ssd.start_time)) > 8
ORDER BY HOUR(TIMEDIFF(ssd.end_time, ssd.start_time)) DESC;
```

Explanation-
In this query, we select the 'staff_ID', 'branch_ID', 'branch_name' and 'shift_date' columns from the 'StaffShiftDetails' table. We also calculate the 'ShiftDuration' by using the 'HOUR(TIMEDIFF())' functions to calculate the difference in hours between the shift end time and the shift start times. We then join the 'StaffShiftDetails' table with the 'BranchDetails' table on the 'branch_ID' column. We then filter the rows based on the shiftduration greater than 8 hours and sort by the shift duration in descending order.

Result:



staff_ID	branch_ID	branch_name	shift_date	ShiftDuration
48	24	Branch 24	2022-06-02	11
12	21	Branch 21	2022-08-21	10
24	37	Branch 37	2022-05-31	10
32	37	Branch 37	2022-08-14	10
49	31	Branch 31	2022-01-04	10
1	15	Branch 15	2022-12-28	9
16	36	Branch 36	2022-12-08	9

Staff 48 at branch 24 had the longest shift duration of 11 hours on 2nd June 2022. This table will allow AussieRetailers to roster staff with equal working hours instead of giving a couple of employees too long working durations.

Q 15. Find the product with the highest number of complaints and detail the nature of these complaints.

```
WITH ProductWithMaxComplaints AS (
    SELECT cod.product_ID, count(cod.complaint_ID) AS ComplaintCount
    FROM complaintdetails cod
    GROUP BY cod.product_id
    ORDER BY count(cod.complaint_ID) DESC
```

```
            LIMIT 1
    ),
    ProductFeatures AS (
            SELECT  pwmc.product_ID, pd.name AS product_name, cd.complaint_ID,
            cd.complaint_date, cd.resolution_status
            FROM ProductWithMaxComplaints pwmc
            JOIN ComplaintDetails cd ON pwmc.product_ID = cd.product_ID
            JOIN ProductDetails pd ON cd.product_ID = pd.product_ID
    )
    SELECT product_ID, product_name, complaint_ID, complaint_date, resolution_status
    FROM ProductFeatures;
```

Explanation-

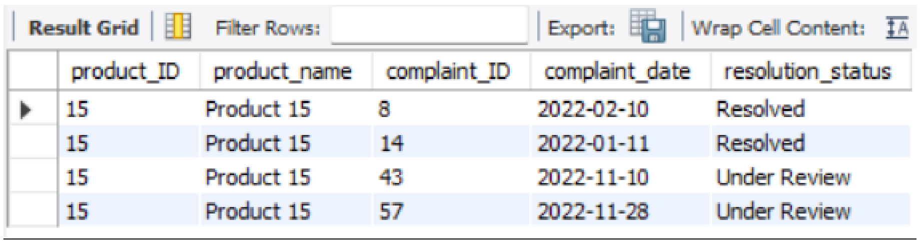
This query involves two CTEs: ‘ProductWithMaxComplaints’ and ‘ProductFeatures’.

ProductWithMaxFeatures CTE: This CTE, computes the product with the max number of complaints. To do this, we select the ‘product_ID’ and the count of complaints using the COUNT() function. Then, we group by the ‘product_ID’ column and sort the rows in descending order of ‘ComplaintCount’. Finally, we use the LIMIT clause to select the top product with the most complaints.

ProductFeature CTE: In this subquery, we select the relevant columns by joining the ‘ComplaintDetails’ table with the ‘ProductWithMaxComplaints’ CTE defined above.

Finally, we select the important features of that product like ‘product_ID’, ‘product_name’, ‘complaint_ID’, ‘complaint_date’, and ‘resolution_status’ from the ‘ProductFeature’ CTE.

Result:



product_ID	product_name	complaint_ID	complaint_date	resolution_status
15	Product 15	8	2022-02-10	Resolved
15	Product 15	14	2022-01-11	Resolved
15	Product 15	43	2022-11-10	Under Review
15	Product 15	57	2022-11-28	Under Review

From the results we can see that Product 15 was the product with the most complaints at 4 complaints. Of those, 2 were resolved and 2 are still under review. AussieRetailers can use this information to understand why so many customers have issues with Product 15 and can work towards improving the customer experience.

Self-Made Questions

Q 16. Which branches have the highest average sales per transaction over the past year?

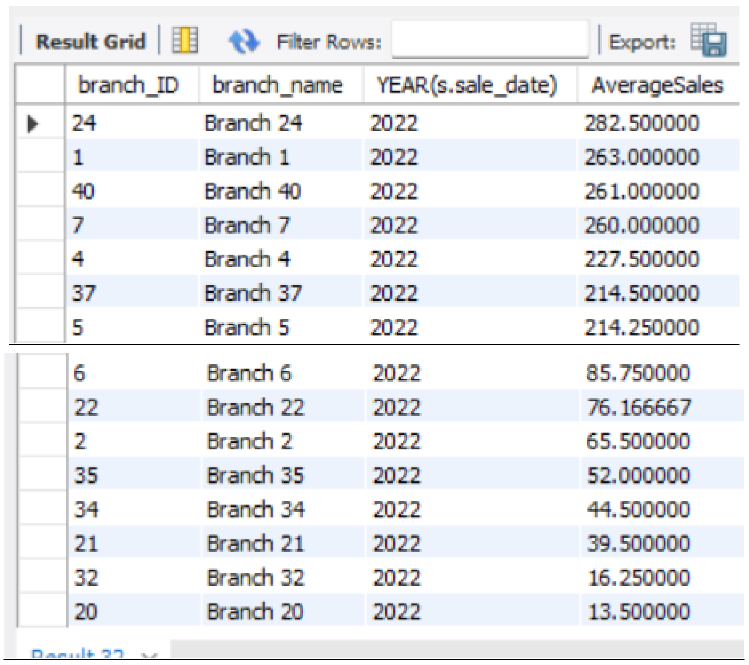
Reason: The reason I chose this question is because identifying branches with the highest average sales per transaction can help in understanding regional sales performance and customer purchasing behaviour. This insight can be used to replicate successful strategies in other branches to boost their performance as well.

```
    SELECT b.branch_ID, b.branch_name, YEAR(s.sale_date), AVG(s.quantity * pd.price) AS
    AverageSales
    FROM branchdetails b
```

```
JOIN Sales s ON s.branch_ID = b.branch_id
JOIN productdetails pd ON pd.product_id = s.product_ID
GROUP BY b.branch_ID, b.branch_name, YEAR(s.sale_date)
ORDER BY AverageSales DESC;
```

Explanation-
In this query, we first select the ‘branch_ID’ and ‘branch_name’ from the ‘BranchDetails’ table. Then we select the year of each sale using the ‘YEAR()’ function and calculate the ‘AverageSales’ (alias name) by using the ‘AVG(s.quantity *pd.price)’ aggregate function. We then join the ‘Sales’, ‘ProductDetails’ and ‘BranchDetails’ tables together. WE finally group by the ‘branch_ID’, ‘branch_name’ and the sale year and sort the rows in descending order of sales to get the desired result.

Result:



branch_ID	branch_name	YEAR(s.sale_date)	AverageSales
24	Branch 24	2022	282.500000
1	Branch 1	2022	263.000000
40	Branch 40	2022	261.000000
7	Branch 7	2022	260.000000
4	Branch 4	2022	227.500000
37	Branch 37	2022	214.500000
5	Branch 5	2022	214.250000
6	Branch 6	2022	85.750000
22	Branch 22	2022	76.166667
2	Branch 2	2022	65.500000
35	Branch 35	2022	52.000000
34	Branch 34	2022	44.500000
21	Branch 21	2022	39.500000
32	Branch 32	2022	16.250000
20	Branch 20	2022	13.500000

This is just a snippet of the results table. Branch 24 performed the best in the 2022 in terms of average sales as it generated the highest average sales of $282.5. Meanwhile, Branch 20 was the lowest performing branch as it generated just an average sale of $13.5.

Q 17. Determine the seasonal (monthly) sales trends for different product categories

Reason: Identifying seasonal trends can allow AussieRetailers to anticipate demand fluctuations and adjust inventory levels accordingly. This can help in reducing stockout and overstocking situations, thus optimizing inventory costs and improving customer satisfaction.

```
WITH MonthlySales AS (
        SELECT DATE_FORMAT(s.sale_date, '%Y-%m') AS sale_month, p.category,
        SUM(s.quantity * p.price) AS total_sales_amount
        FROM Sales s
        JOIN ProductDetails p ON s.product_ID = p.product_ID
        GROUP BY sale_month, p.category
)
SELECT category, sale_month, SUM(total_sales_amount) AS total_sales
FROM MonthlySales
```

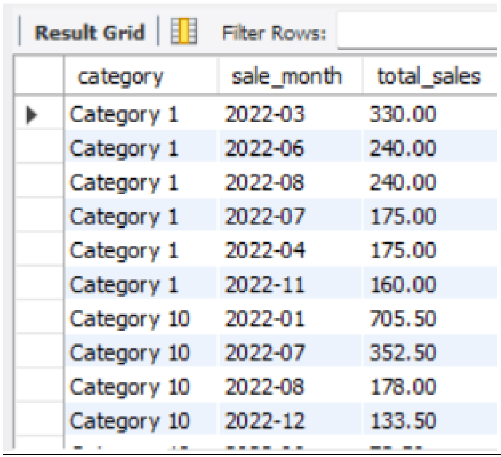
```sql
GROUP BY category, sale_month
ORDER BY category, total_sales DESC;
```

Explanation-
In order to solve this question, I first aggregated the sales data by month and product category. Then I calculated the sum total sales amount for each category per month and then grouped the results by month and category to see the trends. This query required the use of CTE:

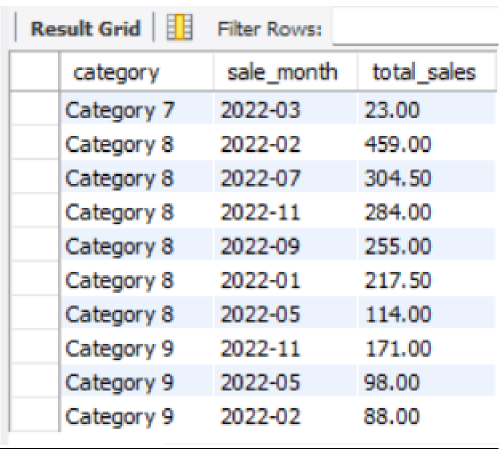
MonthlySales CTE: This subquery calculates the total sales amount per product for each category. This is done by first selecting the sale month. I did this by using the DATE_FORMAT() function and changing the format of the ‘sale_date’ to ‘YY-mm’. Then I calculated the ‘SUM()’ aggregate of the product of ‘quantity’ and ‘price’ to get the ‘total_sales_amount’. Then I joined the ‘Sales’ and ‘ProductDetails’ tables and grouped the rows by the ‘sale_month’ and ‘category’ columns.

The final SELECT statement aggregates the results from the ‘MonthlySales’ CTE to show the total sales amount per category for each month. This is done by grouping the relevant columns by the ‘category’ and ‘sale_month’ columns.

Results:



category	sale_month	total_sales
Category 1	2022-03	330.00
Category 1	2022-06	240.00
Category 1	2022-08	240.00
Category 1	2022-07	175.00
Category 1	2022-04	175.00
Category 1	2022-11	160.00
Category 10	2022-01	705.50
Category 10	2022-07	352.50
Category 10	2022-08	178.00
Category 10	2022-12	133.50



category	sale_month	total_sales
Category 7	2022-03	23.00
Category 8	2022-02	459.00
Category 8	2022-07	304.50
Category 8	2022-11	284.00
Category 8	2022-09	255.00
Category 8	2022-01	217.50
Category 8	2022-05	114.00
Category 9	2022-11	171.00
Category 9	2022-05	98.00
Category 9	2022-02	88.00

This is a snippet of the results table. We can see each category’s monthly (seasonal) sales. Category had its highest sales in March 2022, while it had its lowest sales in November 2022. On the other hand, Category 8 had its highest sales in February 2022 and its lowest sales in May 2022. This information can be crucial for AussieRetailers to understand consumer trends and to stock items accordingly.

Q 18. Identify the top 5 branches with the highest stock levels and compare their sales performance.
Reason: Comparing stock levels with sales performance can highlight efficiency in inventory management. While high stock levels with low sales might indicate overstocking, low stock level with high sales may indicate missed sales opportunities due to understocking.

```sql
WITH TopBranches AS (
    SELECT s.branch_ID, b.branch_name, SUM(pd.stock_level) AS total_stock_level
```

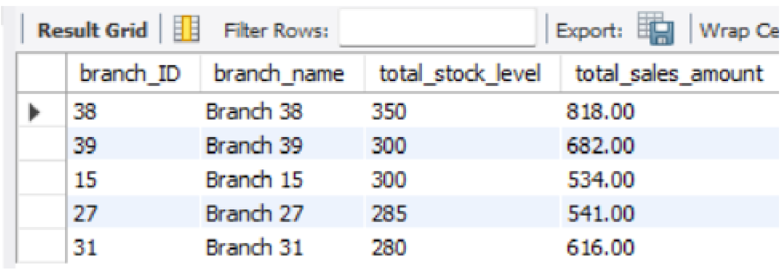
```
        FROM Sales s
        JOIN ProductDetails pd ON s.product_ID = pd.product_ID
        JOIN BranchDetails b ON b.branch_ID = s.branch_ID
        GROUP BY b.branch_ID, b.branch_name
        ORDER BY total_stock_level DESC
        LIMIT 5
    )
    SELECT tb.branch_ID, tb.branch_name, tb.total_stock_level,
    COALESCE(SUM(s.quantity * pd.price), 0) AS total_sales_amount
    FROM TopBranches tb
    LEFT JOIN Sales s ON tb.branch_ID = s.branch_ID
    LEFT JOIN ProductDetails pd ON s.product_ID = pd.product_ID
    GROUP BY tb.branch_ID, tb.branch_name, tb.total_stock_level
    ORDER BY tb.total_stock_level DESC;
```

Explanation-
In order to solve this question, I first calculated the stock levels for each branch. Then I identified the top 5 branches with the highest stock levels. Nest, I computed the sales performance for these top 5 branches and combined the stock levels and sales performances for comparison. This query required the use of CTE:

TopBranches CTE: This CTE selects the top 5 branches in terms of total stock levels. To perform this, I used the SUM() aggregate function to calculate the 'total_stock_level'. I then joined the relevant tables using the JOIN() clause and then grouped the results by 'branch_ID' and 'branch_name'. Finally, I order the results in descending order of total stock level and used the LIMIT() clause to select the top 5 branches.

The final SELECT statement, joins the 'TopBranches' CTE with the 'Sales' and 'ProductDetails' tables to calculate the sales amount for each of the top 5 branches. The reason I have used 'LEFT JOIN()' here is to ensure that all branches identified in the 'TopBranches' CTE are included in the final result set, even if they don't have any corresponding sales records in the 'Sales' table. This ensures that the top branches by stock levels are not excluded from the results if they haven't recorded any sales transactions. The 'COALESCE' function ensures the branches without sales still appear in the results with a total sales amount of 0. The results are then grouped by branch and order by their total stock level to get the desired result.

Result:



branch_ID	branch_name	total_stock_level	total_sales_amount
38	Branch 38	350	818.00
39	Branch 39	300	682.00
15	Branch 15	300	534.00
27	Branch 27	285	541.00
31	Branch 31	280	616.00

Branch 38 had the highest stock level of 350 units and brought in the highest sales value of $818. Meanwhile branch 31 was placed fifth in terms of total stock level. However, it ranked third overall in terms of total sales amount as it brought in a sales value of $616.

Q 19. Determine the average shift duration for staff across different branches over the last 6 months

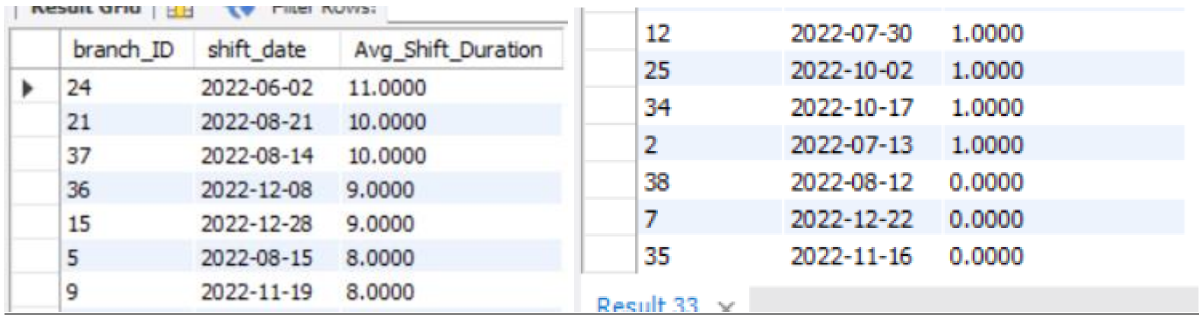
Reason: Analyzing the average shift durations for staff across different branches can help in understanding staffing patterns and potential overwork or underutilization of staff. This can inform more efficient shift scheduling and workforce management.

```
SELECT ssd.branch_ID, ssd.shift_date, AVG(HOUR(TIMEDIFF(ssd.end_time,
ssd.start_time))) AS Avg_Shift_Duration
FROM staffshiftdetails ssd
WHERE ssd.shift_date BETWEEN "2022-06-01" and "2022-12-31"
GROUP BY ssd.branch_id, ssd.shift_date
ORDER BY Avg_Shift_Duration DESC;
```

Explanation-

We first select the 'branch_ID' and 'shift_date' columns from the 'StaffShiftDetails' table. Then we calculate the 'Avg_Shift_Duration' (alias name) by using the 'AVG(HOUR(TIMEDIFF(ssd.end_time, ssd.start_time)))' function. We then use the 'WHERE()' clause to filter the rows based on the shift dates that occurred over the past 6 months. Finally, we grouped the data by the 'branch_ID' and the 'shift_date' and sorted the rows in descending order of average shift duration.

Results:



branch_ID	shift_date	Avg_Shift_Duration
24	2022-06-02	11.0000
21	2022-08-21	10.0000
37	2022-08-14	10.0000
36	2022-12-08	9.0000
15	2022-12-28	9.0000
5	2022-08-15	8.0000
9	2022-11-19	8.0000
12	2022-07-30	1.0000
25	2022-10-02	1.0000
34	2022-10-17	1.0000
2	2022-07-13	1.0000
38	2022-08-12	0.0000
7	2022-12-22	0.0000
35	2022-11-16	0.0000

This is a snippet of the results table. We can clearly see that branch 24 has the highest average shift duration at 11 hours, followed by branches 21 and 37 at 10 hours each. On the other hand there are 3 branches (Branches 38,7 and 35) which have an average shift duration of 0 hours. This clearly shows that these branches have no staff working there. This should ring alarms at AussieRetailers as there is a huge gap in the average shift durations between branches. They must improve their staffing patterns and utilize their resources more efficiently.

Q 20. Determine the number of customers that have opted for a loyalty program. Also calculate the total and average customer spending based on the loyalty program status.

Reason: By analyzing spending habits of customers based on the loyalty program status, AussieRetailers can evaluate the effectiveness of its loyalty program and tailor promotions or rewards to different customer segments.

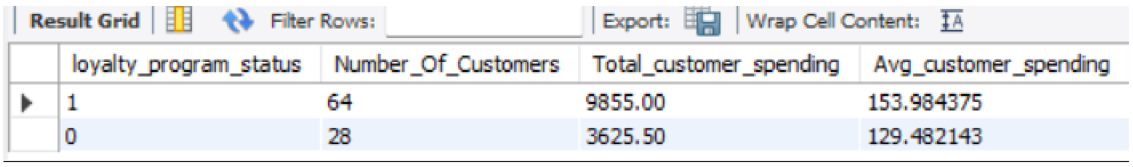
```
SELECT cd.loyalty_program_status, count(cd.loyalty_program_status) AS
Number_Of_Customers,  SUM(s.quantity * pd.price) AS Total_customer_spending,
AVG(s.quantity * pd.price) AS Avg_customer_spending
```

```
FROM customerdetails cd
JOIN sales s ON cd.customer_ID = s.customer_ID
JOIN productdetails pd ON pd.product_id = s.product_id
GROUP BY cd.loyalty_program_status;
```

Explanation-

We first select the ‘loyalty_program_status’ from the ‘CustomerDetails’ table. Then we calculate the ‘Number_Of_Customers’ (alias name) using the ‘COUNT()’ function and the ‘Total_customer_spending’ (alias name) using the ‘SUM()’ function. We then join the ‘CustomerDetails’ , ‘Sales’, and ‘ProductDetails’ tables using the ‘JOIN()’ clause on the appropriate columns. Finally, we group the data by the ‘loyalty_program_status’ column to get the required column.

Results:



loyalty_program_status	Number_Of_Customers	Total_customer_spending	Avg_customer_spending
1	64	9855.00	153.984375
0	28	3625.50	129.482143

We can clearly see that almost two thirds of customers have a loyalty program status (assuming 1 means yes) and a third of customers don’t. We can also clearly see that the total customer spending and the average customer spending is higher in customers with a loyalty program status. As a result, AussieRetailers’ aim should be to get all its customers signed up for its loyalty program status as that may lead to greater sales and profitability.