

SQL Analytics Projects

MavenMovies & ClassicModels Databases

Author: Vishnu Vardhan Reddy Kukudala

Role: Aspiring Data Scientist / Data Analyst

Section 1: SQL Analysis on MavenMovies Database

Q1. Monthly Rental Trends

Objective & Explanation:

This analysis examines how movie rentals vary across different months to identify seasonal demand patterns. By aggregating rentals at the monthly level, we can observe peaks and dips in customer activity over time. These trends help management understand high-demand periods and plan inventory, promotions, and staffing accordingly.

The screenshot shows the MySQL Workbench interface with two tabs: 'mavenmovies_queries' and 'classicmodels_queries'. The 'mavenmovies_queries' tab is active, displaying the following SQL code:

```
1 -- Q1: Monthly Rental Trends
2 -- Purpose: Analyze how rentals vary across months to identify seasonality
3
4 • SELECT
5     DATE_FORMAT(rental_date, '%Y-%m') AS rental_month,
6     COUNT(*) AS total_rentals
7     FROM rental
8     GROUP BY rental_month
9     ORDER BY rental_month;
10
11 -- Q2: Peak Rental Hours
12 -- Purpose: Identify the hours of the day with highest rental activity.
```

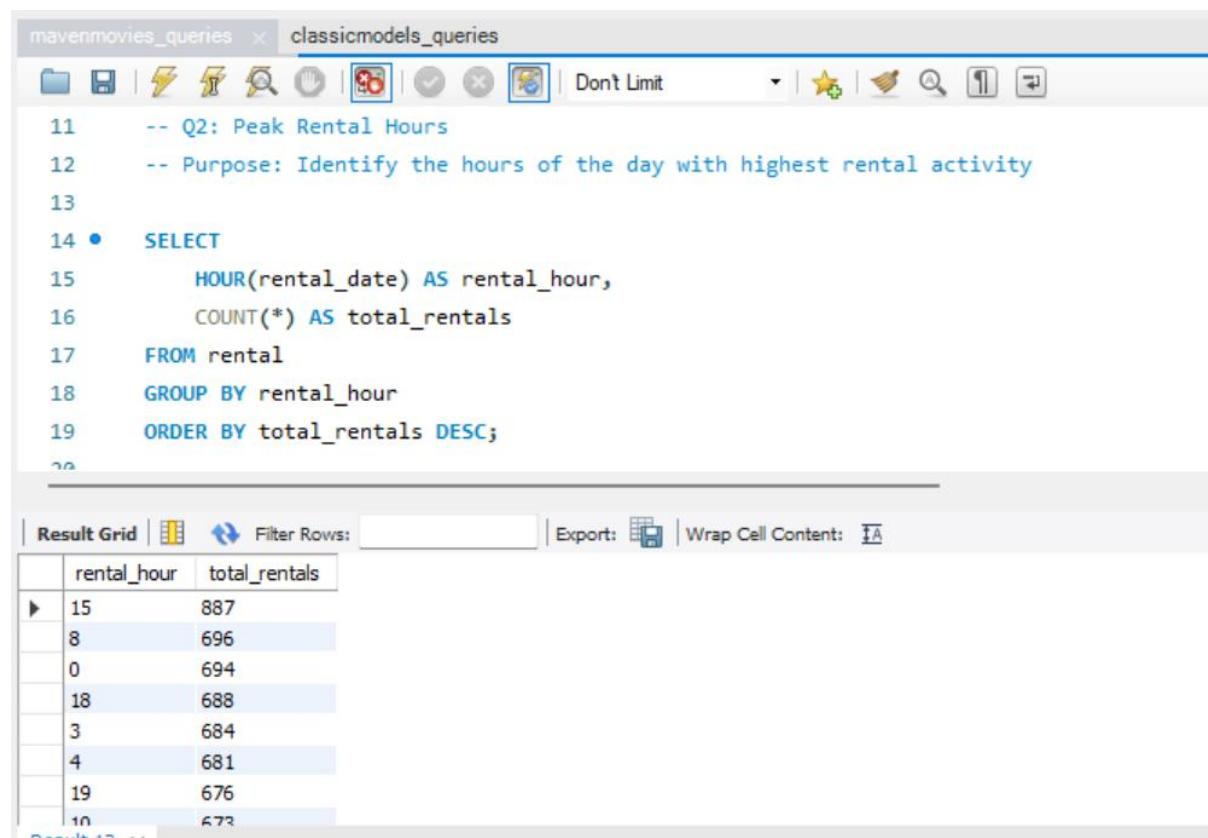
Below the code, the 'Result Grid' shows the following data:

rental_month	total_rentals
2005-05	1156
2005-06	2311
2005-07	6709
2005-08	5686
2006-02	182

Q2. Peak Rental Hours

Objective & Explanation:

This query identifies the hours of the day during which rental activity is highest. Understanding peak rental hours allows the business to allocate staff more efficiently and ensure smooth operations during busy periods. It also provides insights into customer behaviour and preferred rental times.



The screenshot shows a database query interface with the following details:

- Query Text:**

```
11      -- Q2: Peak Rental Hours
12      -- Purpose: Identify the hours of the day with highest rental activity
13
14 •  SELECT
15      HOUR(rental_date) AS rental_hour,
16      COUNT(*) AS total_rentals
17  FROM rental
18  GROUP BY rental_hour
19  ORDER BY total_rentals DESC;
```
- Result Grid:**

rental_hour	total_rentals
15	887
8	696
0	694
18	688
3	684
4	681
19	676
10	673

Q3. Top 10 Most Rented Films

Objective & Explanation:

This analysis identifies the most frequently rented films based on total rental count. Highlighting popular titles helps in optimizing inventory decisions and ensuring high-demand movies are always available. It also provides insights into customer preferences and content popularity.

The screenshot shows a MySQL Workbench interface with two tabs: 'mavenmovies_queries' and 'classicmodels_queries'. The 'classicmodels_queries' tab is active, displaying a SQL query to find the top 10 most rented films. The query uses joins between 'film', 'inventory', and 'rental' tables to group by film title and count the rentals. The results grid shows the top 6 films, all of which have a rental count of 32.

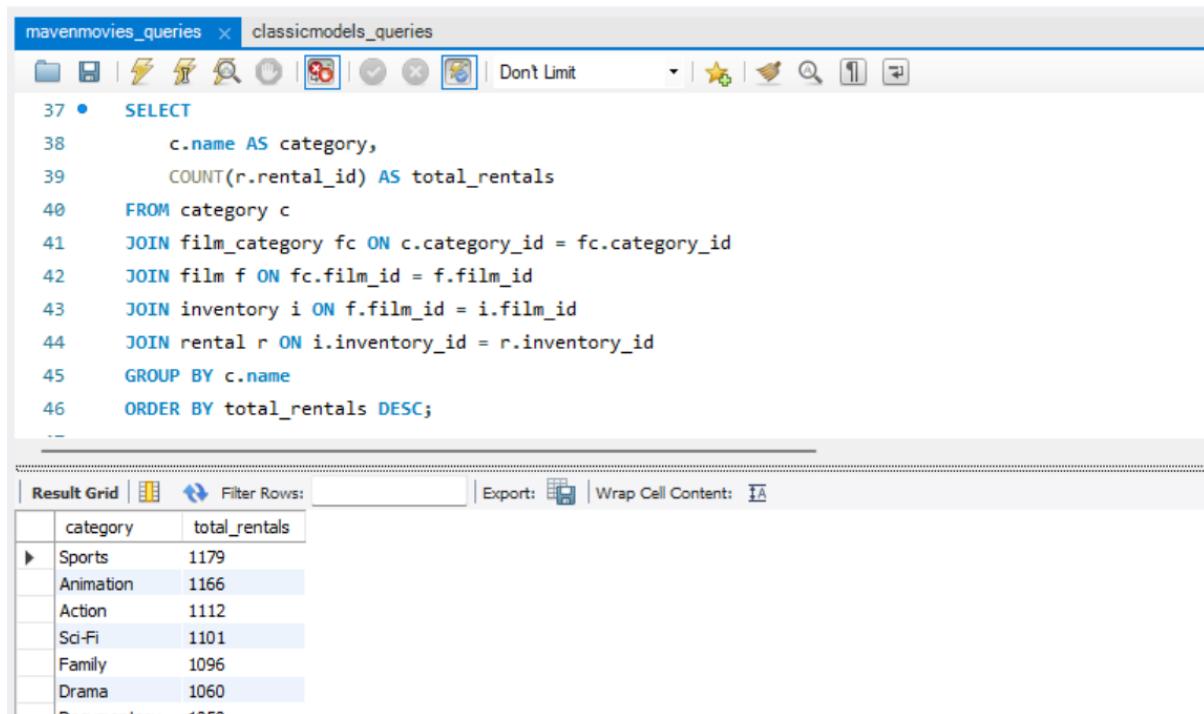
```
23
24 •  SELECT
25      f.title AS film_title,
26      COUNT(r.rental_id) AS rental_count
27  FROM film f
28  JOIN inventory i ON f.film_id = i.film_id
29  JOIN rental r ON i.inventory_id = r.inventory_id
30  GROUP BY f.title
31  ORDER BY rental_count DESC
32  LIMIT 10;
33
```

film_title	rental_count
BUCKET BROTHERHOOD	34
ROCKETEER MOTHER	33
RIDGEMONT SUBMARINE	32
GRIT CLOCKWORK	32
SCALAWAG DUCK	32
JUGGLER HARDLY	32

Q4. Film Categories with Highest Rentals

Objective & Explanation:

This query evaluates rental demand across different film categories to determine which genres are most popular. Understanding category-level demand helps in targeted content acquisition and marketing strategies. It also supports data-driven decisions related to genre-focused promotions.



The screenshot shows a database query interface with the following details:

Query Editor:

```
37 •   SELECT
38     c.name AS category,
39     COUNT(r.rental_id) AS total_rentals
40   FROM category c
41   JOIN film_category fc ON c.category_id = fc.category_id
42   JOIN film f ON fc.film_id = f.film_id
43   JOIN inventory i ON f.film_id = i.film_id
44   JOIN rental r ON i.inventory_id = r.inventory_id
45   GROUP BY c.name
46   ORDER BY total_rentals DESC;
```

Result Grid:

category	total_rentals
Sports	1179
Animation	1166
Action	1112
Sci-Fi	1101
Family	1096
Drama	1060
Thriller	1050

Q5. Store with Highest Rental Revenue

Objective & Explanation:

This analysis calculates total rental revenue generated by each store to identify the top-performing location. Comparing store-level revenue helps management evaluate operational efficiency and profitability. These insights can guide strategic decisions such as resource allocation and performance benchmarking.

The screenshot shows a MySQL Workbench interface with two tabs: "mavenmovies_queries" and "classicmodels_queries". The "classicmodels_queries" tab is active, displaying a SQL query and its results. The query is as follows:

```
50
51 •  SELECT
52      s.store_id,
53      SUM(p.amount) AS total_revenue
54  FROM store s
55  JOIN staff st ON s.store_id = st.store_id
56  JOIN payment p ON st.staff_id = p.staff_id
57  GROUP BY s.store_id
58  ORDER BY total_revenue DESC;
59
```

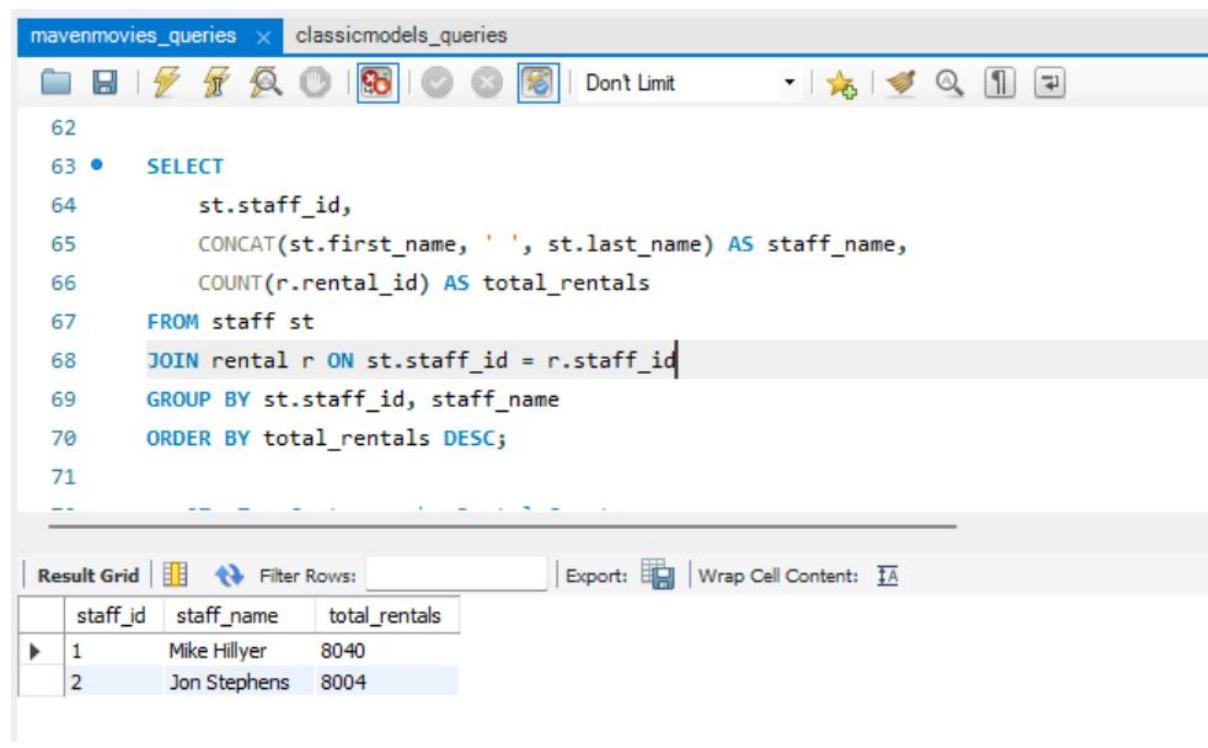
The results grid shows the following data:

	store_id	total_revenue
▶	2	33927.04
	1	33489.47

Q6. Rental Distribution by Staff Members

Objective & Explanation:

This query measures staff performance by analyzing the number of rentals processed by each staff member. It helps identify high-performing employees and assess workload distribution. Such insights can support staffing decisions, training needs, and performance evaluations.



The screenshot shows a database query editor interface with the following details:

- Query Tab:** The tab is titled "classicmodels_queries".
- Toolbar:** Includes standard icons for file operations, search, and navigation.
- Query Editor:** Displays the SQL code for the query:

```
62
63 •  SELECT
64      st.staff_id,
65      CONCAT(st.first_name, ' ', st.last_name) AS staff_name,
66      COUNT(r.rental_id) AS total_rentals
67  FROM staff st
68  JOIN rental r ON st.staff_id = r.staff_id
69  GROUP BY st.staff_id, staff_name
70  ORDER BY total_rentals DESC;
71
```
- Result Grid:** A table showing the results of the query:

	staff_id	staff_name	total_rentals
▶	1	Mike Hillyer	8040
	2	Jon Stephens	8004

Q7. Top Customers by Rental Count

Objective & Explanation:

This analysis identifies the most loyal customers based on their rental frequency. Recognizing high-engagement customers enables the business to design loyalty programs and personalized offers. It also helps in understanding customer retention and long-term value.

The screenshot shows a MySQL Workbench interface with two tabs: 'mavenmovies_queries' and 'classicmodels_queries'. The 'classicmodels_queries' tab is active, displaying a SQL query to find the top 10 customers by rental count. The query uses JOIN, GROUP BY, and ORDER BY clauses. The results are displayed in a grid below, showing customer IDs and names along with their rental counts.

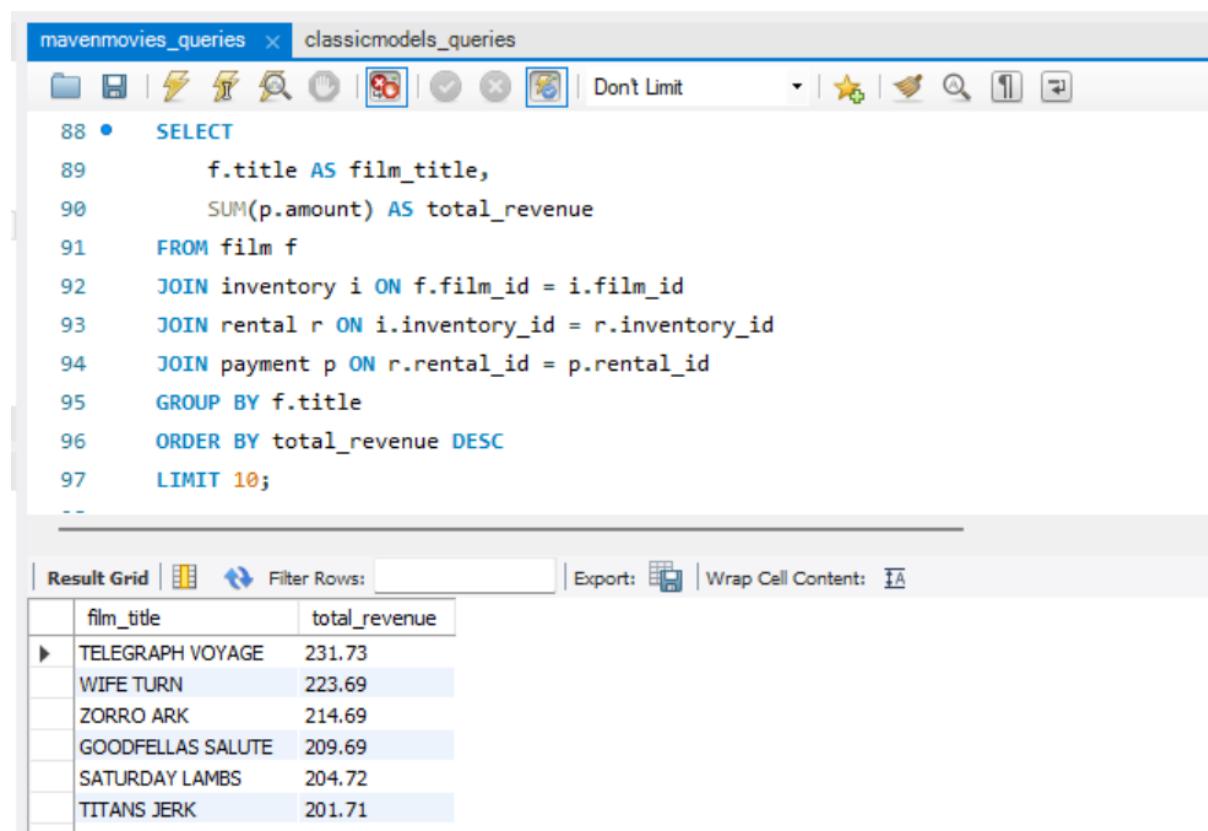
```
74
75 •  SELECT
76     c.customer_id,
77     CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
78     COUNT(r.rental_id) AS rental_count
79     FROM customer c
80     JOIN rental r ON c.customer_id = r.customer_id
81     GROUP BY c.customer_id, customer_name
82     ORDER BY rental_count DESC
83     LIMIT 10;
```

	customer_id	customer_name	rental_count
▶	148	ELEANOR HUNT	46
	526	KARL SEAL	45
	144	CLARA SHAW	42
	236	MARCIA DEAN	42
	75	TAMMY SANDERS	41
	197	SUE PETERS	40
	460	WESLEY DILL	40

Q8. Films Generating the Highest Revenue

Objective & Explanation:

This query identifies films that contribute the most to overall revenue, not just rental volume. It highlights cases where fewer rentals still generate higher income due to pricing factors. These insights help in pricing strategy evaluation and revenue optimization.



The screenshot shows a database query editor interface with two tabs: "mavenmovies_queries" and "classicmodels_queries". The "classicmodels_queries" tab is active, displaying a SQL query and its results. The query retrieves the top 10 films based on total revenue. The results are presented in a grid format.

```
88 •    SELECT
89      f.title AS film_title,
90      SUM(p.amount) AS total_revenue
91  FROM film f
92  JOIN inventory i ON f.film_id = i.film_id
93  JOIN rental r ON i.inventory_id = r.inventory_id
94  JOIN payment p ON r.rental_id = p.rental_id
95  GROUP BY f.title
96  ORDER BY total_revenue DESC
97  LIMIT 10;
```

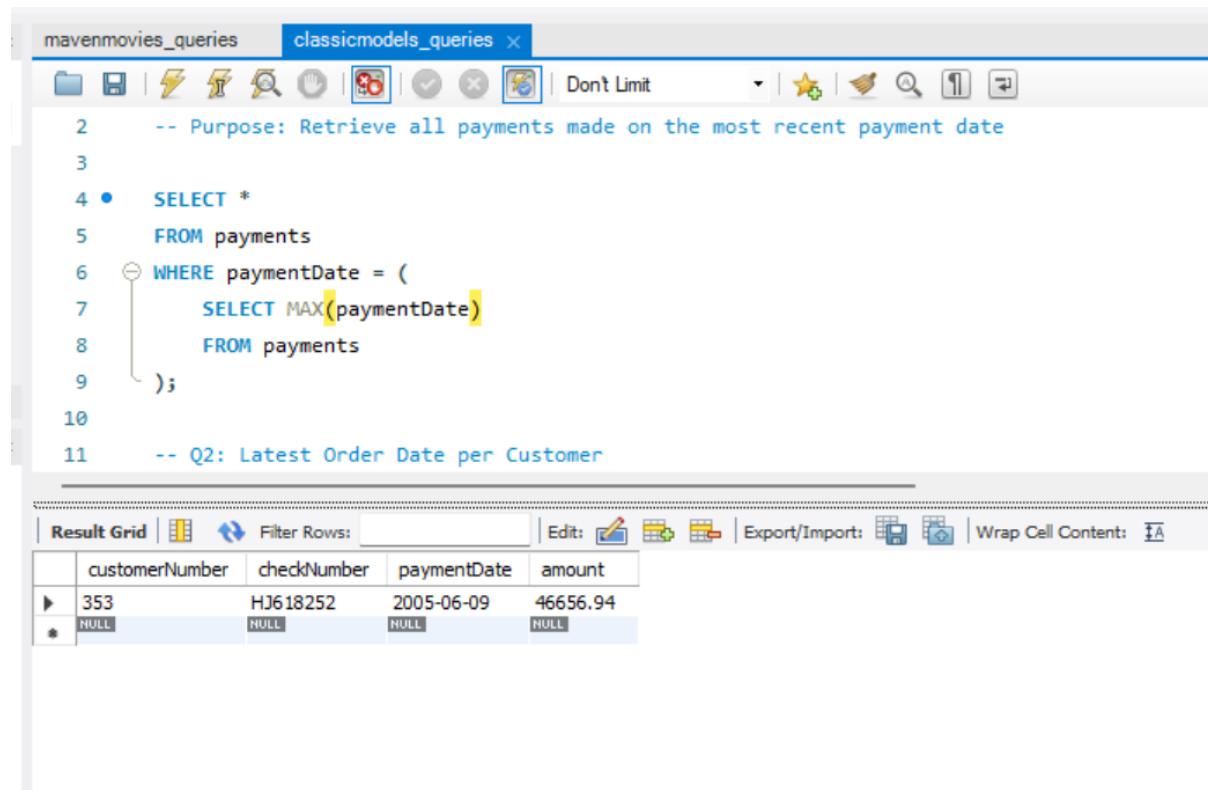
film_title	total_revenue
TELEGRAPH VOYAGE	231.73
WIFE TURN	223.69
ZORRO ARK	214.69
GOODFELLAS SALUTE	209.69
SATURDAY LAMBS	204.72
TITANS JERK	201.71
TOPOLIE ROUND	199.72

Section 2: Advanced SQL Analytics (ClassicModels)

Q1. Payments on the Latest Payment Date

Objective & Explanation:

This analysis retrieves all payments that occurred on the most recent payment date in the database. It helps identify the latest financial activity and ensures accurate reporting of recent transactions. Such queries are useful for end-of-day or period-close financial analysis.



The screenshot shows a SQL query editor interface with two tabs: "mavenmovies_queries" and "classicmodels_queries". The "classicmodels_queries" tab is active, showing a query to retrieve the latest payment date from the "payments" table. The query uses a subquery to find the maximum payment date and then selects all rows where the payment date matches this maximum value. The result grid displays one row with customer number 353, check number HJ618252, payment date 2005-06-09, and amount 46656.94.

```
2  -- Purpose: Retrieve all payments made on the most recent payment date
3
4 •  SELECT *
5   FROM payments
6   WHERE paymentDate = (
7     SELECT MAX(paymentDate)
8     FROM payments
9   );
10
11  -- Q2: Latest Order Date per Customer
```

	customerNumber	checkNumber	paymentDate	amount
▶	353	HJ618252	2005-06-09	46656.94
*	HULL	HULL	HULL	HULL

Q2. Latest Order Date per Customer

Objective & Explanation:

This query identifies the most recent order placed by each customer. It provides insights into customer activity and engagement levels. Understanding recent customer interactions helps in targeting active customers and identifying inactive ones.

```
mavenmovies_queries    classicmodels_queries
11      -- Q2: Latest Order Date per Customer
12      -- Purpose: Show each customer with their most recent order date
13
14 •   SELECT
15      customerNumber,
16      MAX(orderDate) AS latest_order_date
17  FROM orders
18  GROUP BY customerNumber;
19
20  -- Q3+ Products with Below-Average Stock
```

customerNumber	latest_order_date
103	2004-11-25
112	2004-11-29
114	2004-11-29
119	2005-05-31
121	2004-11-05
124	2005-05-29
128	2004-11-05

Q3. Products with Below-Average Stock in Their Product Line

Objective & Explanation:

This analysis compares product stock levels against the average stock within the same product line. It helps identify items that may be at risk of stockouts relative to similar products. These insights support inventory planning and replenishment decisions.

The screenshot shows a database query editor interface. The top tab is labeled "classicmodels_queries". The query window displays the following SQL code:

```
23 •   SELECT
24     productCode,
25     productName,
26     productLine,
27     quantityInStock
28   FROM products p
29   WHERE quantityInStock < (
30       SELECT AVG(quantityInStock)
31     FROM products
32   );
```

Below the code is a "Result Grid" table with the following data:

	productCode	productName	productLine	quantityInStock
▶	S10_4757	1972 Alfa Romeo GTA	Classic Cars	3252
	S12_1099	1968 Ford Mustang	Classic Cars	68
	S12_1108	2001 Ferrari Enzo	Classic Cars	3619
	S12_1666	1958 Setra Bus	Trucks and Buses	1579
	S12_3891	1969 Ford Falcon	Classic Cars	1049
	S18_1097	1940 Ford Pickup Truck	Trucks and Buses	2613

Q4. Customers Whose First Order Happened in 2004

Objective & Explanation:

This query identifies customers whose earliest recorded order occurred in the year 2004. It helps analyze customer acquisition during a specific time period. Such insights are useful for cohort analysis and long-term customer behavior studies.

The screenshot shows a database interface with two tabs at the top: "mavenmovies_queries" and "classicmodels_queries". The "classicmodels_queries" tab is active, showing a SQL query. The code is as follows:

```
35      -- Purpose: Identify customers whose earliest order occurred in 2004
36
37 •   SELECT
38     customerNumber,
39     MIN(orderDate) AS first_order_date
40   FROM orders
41   GROUP BY customerNumber
42   HAVING YEAR(first_order_date) = 2004;
43
44   -- Q5: Products Where MSRP Exceeds All Sale Prices
```

Below the code, there is a "Result Grid" section with the following data:

	customerNumber	first_order_date
▶	119	2004-07-23
	157	2004-07-20
	166	2004-02-04
	173	2004-03-10
	177	2004-01-12
	189	2004-02-12

At the bottom left, it says "Result 25 ×". At the bottom right, it says "Output: [dropdown menu]".

Q5. Products Whose MSRP Exceeds All Historical Sale Prices

Objective & Explanation:

This analysis identifies products where the listed MSRP is higher than any price at which the product has ever been sold. It highlights pricing discrepancies and potential overpricing issues. These insights can guide pricing strategy reviews and discount policies.

The screenshot shows a database query interface with two tabs: 'mavenmovies_queries' and 'classicmodels_queries'. The 'classicmodels_queries' tab is active, displaying a SQL query numbered 47. The query selects product code, name, and MSRP from the 'products' table, where the MSRP is greater than all historical sale prices (obtained via a subquery on 'orderdetails').

```
47 •  SELECT
48      p.productCode,
49      p.productName,
50      p.MSRP
51  FROM products p
52  WHERE p.MSRP > ALL (
53      SELECT od.priceEach
54      FROM orderdetails od
55      WHERE od.productCode = p.productCode
56  );
```

The results grid shows the following data:

	productCode	productName	MSRP
▶	S12_1099	1968 Ford Mustang	194.57
	S18_1129	1993 Mazda RX-7	141.54
	S18_1749	1917 Grand Touring Sedan	170.00
	S18_2238	1998 Chrysler Plymouth Prowler	163.73
	S18_2795	1928 Mercedes-Benz SSK	168.75
	S18_2957	1934 Ford V8 Coupe	62.46

Q6. First Order Date per Customer Using CTE

Objective & Explanation:

This query uses a Common Table Expression (CTE) to calculate each customer's first order date in a structured and readable manner. It demonstrates modular query design and improves clarity when performing multi-step analysis. This approach is useful for scalable and maintainable SQL development.

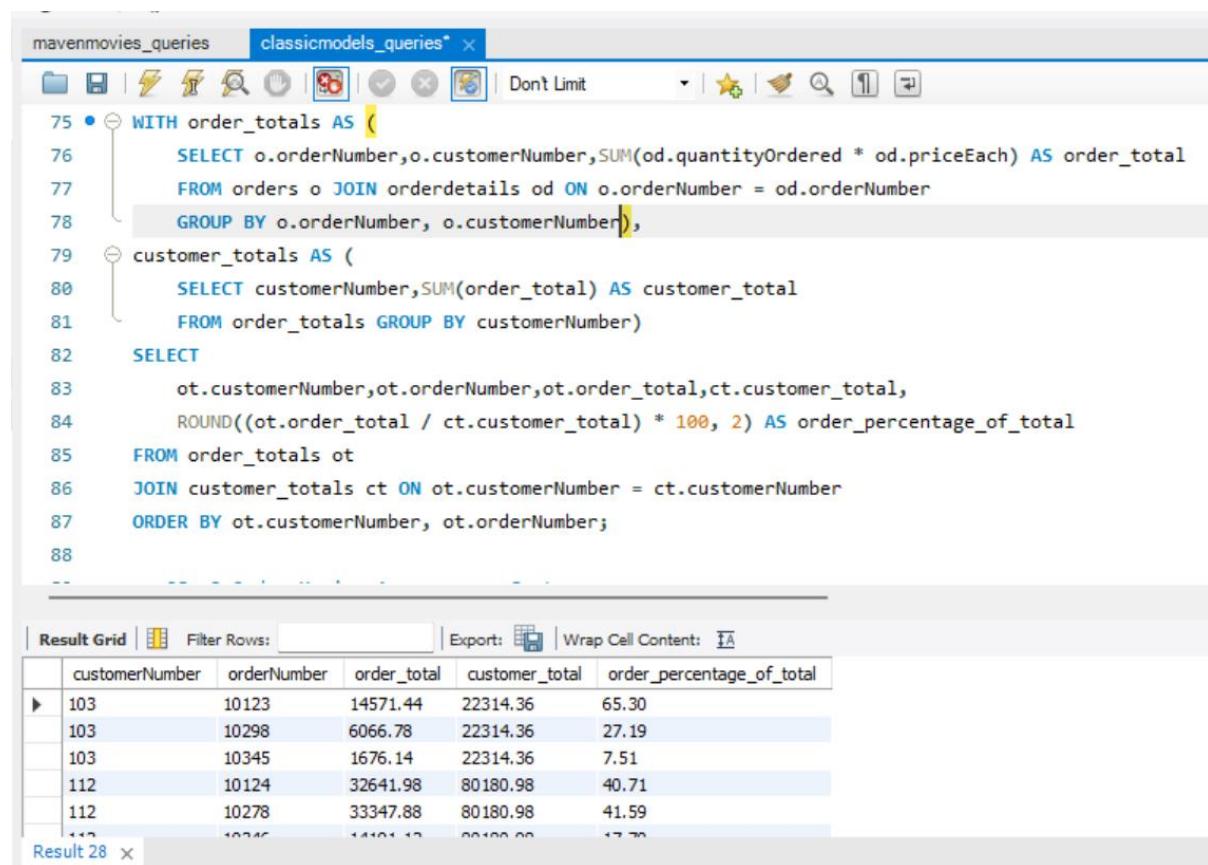
The screenshot shows a database query editor interface with two tabs at the top: "mavenmovies_queries" and "classicmodels_queries". The "classicmodels_queries" tab is active. Below the tabs is a toolbar with various icons for file operations, search, and navigation. The main area displays a SQL script with line numbers 60 through 72. The script defines a CTE named "first_orders" that selects the minimum order date for each customer number from the "orders" table, grouped by customer number. It then selects all columns from this CTE where the year of the first order date is 2004. A comment at the end indicates it's related to Q7: Order Share of Customer Spend. At the bottom, a result grid shows the output of the query, mapping customer numbers to their first order dates. The result grid has a header row with "customerNumber" and "first_order_date". Six rows of data are shown, corresponding to customer numbers 119, 157, 166, 173, 177, and 189, with their respective first order dates.

customerNumber	first_order_date
119	2004-07-23
157	2004-07-20
166	2004-02-04
173	2004-03-10
177	2004-01-12
189	2004-02-12

Q7. Share of Customer Spend by Order (Percentage of Total)

Objective & Explanation:

This analysis calculates each order's contribution to the customer's total spending. It helps identify high-impact orders and spending patterns within individual customers. Such insights are valuable for understanding purchase behavior and revenue concentration.



The screenshot shows a database query editor interface with two tabs: "mavenmovies_queries" and "classicmodels_queries*". The "classicmodels_queries*" tab is active, displaying a multi-step SQL query. The code uses common table expressions (CTEs) to calculate order totals and customer totals, and then joins them to calculate the percentage of total spend for each order. The results are displayed in a grid below the query.

```
75 • WITH order_totals AS (
76     SELECT o.orderNumber, o.customerNumber, SUM(od.quantityOrdered * od.priceEach) AS order_total
77     FROM orders o JOIN orderdetails od ON o.orderNumber = od.orderNumber
78     GROUP BY o.orderNumber, o.customerNumber),
79     customer_totals AS (
80         SELECT customerNumber, SUM(order_total) AS customer_total
81         FROM order_totals GROUP BY customerNumber)
82     SELECT
83         ot.customerNumber, ot.orderNumber, ot.order_total, ct.customer_total,
84         ROUND((ot.order_total / ct.customer_total) * 100, 2) AS order_percentage_of_total
85     FROM order_totals ot
86     JOIN customer_totals ct ON ot.customerNumber = ct.customerNumber
87     ORDER BY ot.customerNumber, ot.orderNumber;
88
```

	customerNumber	orderNumber	order_total	customer_total	order_percentage_of_total
▶	103	10123	14571.44	22314.36	65.30
	103	10298	6066.78	22314.36	27.19
	103	10345	1676.14	22314.36	7.51
	112	10124	32641.98	80180.98	40.71
	112	10278	33347.88	80180.98	41.59
	112	10246	11101.12	80180.98	13.79

Q8. Three-Order Moving Average of Order Value per Customer

Objective & Explanation:

This query computes a rolling average of order values over the current and previous two orders for each customer. It smooths short-term fluctuations and reveals spending trends over time. Moving averages are commonly used in time-series and behavioral analysis.

The screenshot shows a database query editor with the tab 'classicmodels_queries*' selected. The query itself is as follows:

```
92 • Ⓜ WITH order_totals AS (
93     SELECT
94         o.orderNumber, o.customerNumber, o.orderDate,
95         SUM(od.quantityOrdered * od.priceEach) AS order_total
96     FROM orders o
97     JOIN orderdetails od ON o.orderNumber = od.orderNumber
98     GROUP BY o.orderNumber)
99     SELECT
100        customerNumber, orderNumber, orderDate, order_total,
101        ROUND(AVG(order_total)) OVER (
102            PARTITION BY customerNumber
103            ORDER BY orderDate
104            ROWS BETWEEN 2 PRECEDING AND CURRENT ROW ), 2) AS moving_avg_3_orders
105    FROM order_totals ORDER BY customerNumber, orderDate;
```

The result grid displays the following data:

	customerNumber	orderNumber	orderDate	order_total	moving_avg_3_orders
▶	103	10123	2003-05-20	14571.44	14571.44
	103	10298	2004-09-27	6066.78	10319.11
	103	10345	2004-11-25	1676.14	7438.12
	112	10124	2003-05-21	32641.98	32641.98
	112	10278	2004-08-06	33347.88	32994.93
	112	10346	2004-11-29	14191.12	26726.99