RETAIL STORE & INVENTORY MANAGEMENT SYSTEM

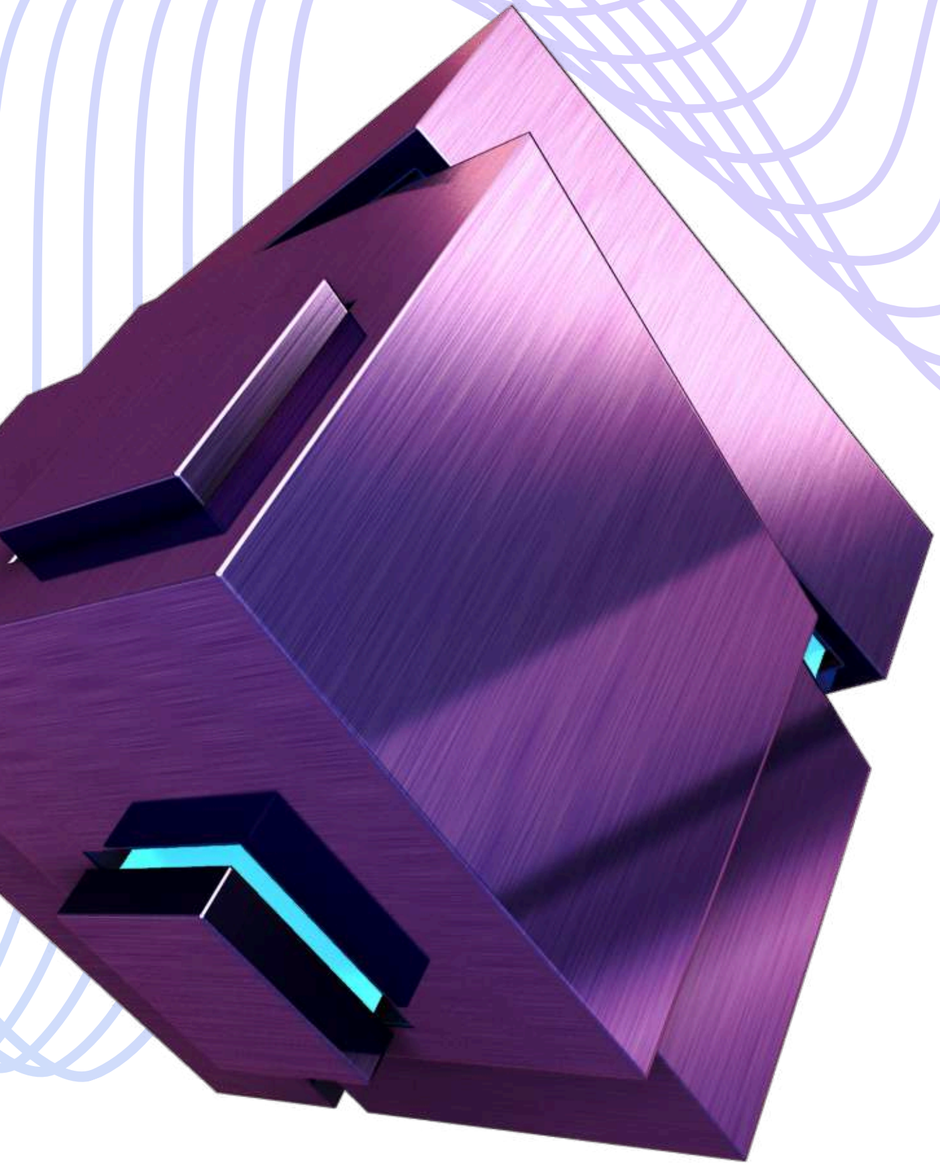
SDU University

Sarsembay Aibat
Yeskaliyev Nurbol
Seiitkhan Zhannur
Nurzhanov Beket

PROJECT GOAL

Objective:

- To create a universal and scalable store and warehouse management system that will allow:
- Efficiently track products, sales and stock
- Analyze customer behavior
- Automate order and return processes



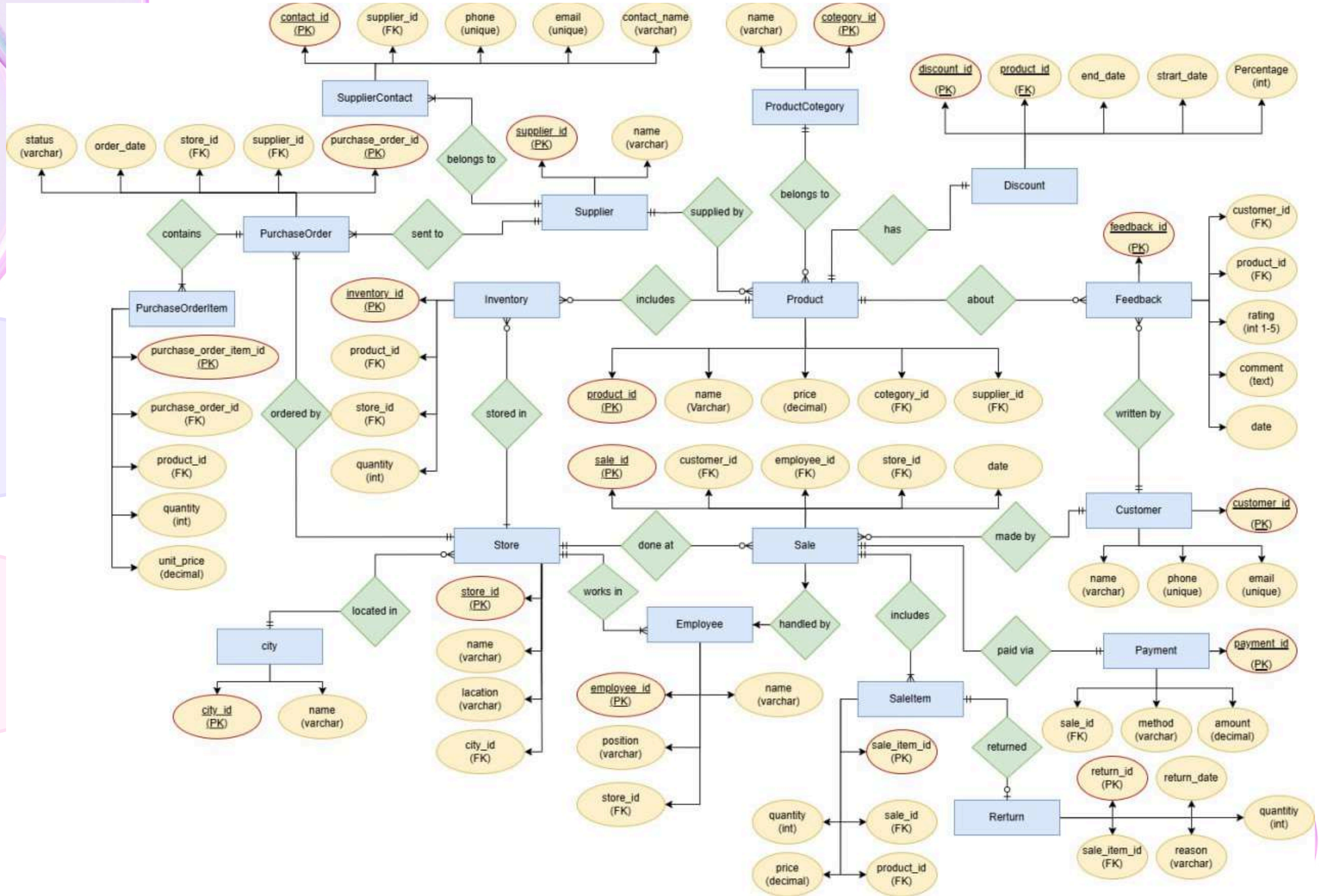
WHY IS SUCH A SYSTEM NEEDED?

Retail business faces problems with inventory, order and personnel management

The lack of a single database leads to errors and loss of profit

The system solves these problems with the help of a centralized database and analytics

ER - DIAGRAM


contact_id (PK)
supplier_id (FK)
phone (unique)
email (unique)
contact_name (varchar)
name (varchar)
cotegory_id (PK)
SupplierContact
ProductCotegory
discount_id (PK)
product_id (FK)
end_date
strart_date
Percentage (int)
status (varchar)
order_date
store_id (FK)
supplier_id (FK)
purchase_order_id (PK)
belongs to
supplier_id (PK)
name (varchar)
Discount
Supplier
supplied by
belongs to
has
contains
PurchaseOrder
sent to
feedback_id (PK)
customer_id (FK)
product_id (FK)
rating (int 1-5)
comment (text)
date
PurchaseOrderItem
inventory_id (PK)
Inventory
includes
Product
about
Feedback
purchase_order_item_id (PK)
product_id (FK)
purchase_order_id (FK)
ordered by
store_id (FK)
stored in
product_id (PK)
name (Varchar)
price (decimal)
cotegory_id (FK)
supplier_id (FK)
written by
product_id (FK)
quantity (int)
quantity (int)
sale_id (PK)
customer_id (FK)
employee_id (FK)
store_id (FK)
date
Customer
customer_id (PK)
unit_price (decimal)
Store
done at
Sale
made by
name (varchar)
phone (unique)
email (unique)
located in
store_id (PK)
works in
Employee
handled by
includes
paid via
Payment
payment_id (PK)
city
name (varchar)
lacation (varchar)
city_id (FK)
employee_id (PK)
name (varchar)
SaleItem
sale_id (FK)
method (varchar)
amount (decimal)
city_id (PK)
name (varchar)
position (varchar)
sale_item_id (PK)
returned
return_id (PK)
return_date
store_id (FK)
quantity (int)
sale_id (FK)
Rerturn
quantity (int)
price (decimal)
product_id (FK)
sale_item_id (FK)
reason (varchar)

NORMALIZATION FORM

Result:

- Removed duplicate data groups
- No transitive dependencies
- All dependencies are from the primary key only

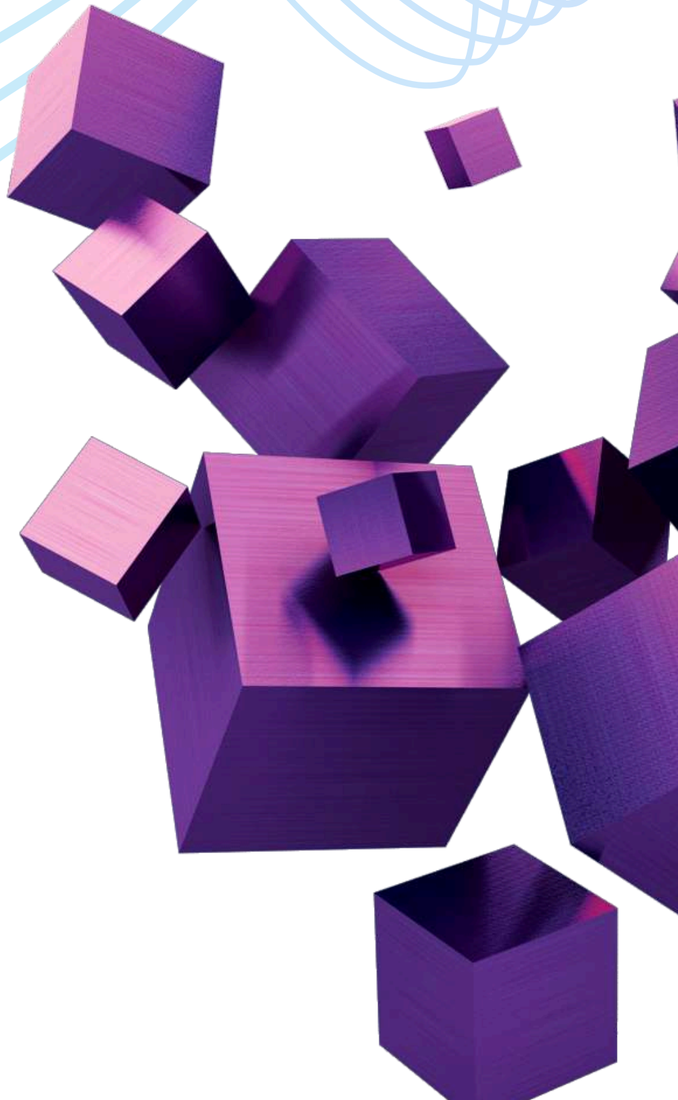
DATA ANALYTICS AND STATISTICS

STORE LIST WITH ADDRESSES

```sql
SELECT s.store_id, s.name, s.street_number, s.street_name, c.name AS city_name
FROM STORE_M s
JOIN CITY_M c ON s.city_id = c.city_id;
```

Results | Explain | Describe | Saved SQL | History

STORE_ID	NAME	STREET_NUMBER	STREET_NAME	CITY_NAME
1	Hermann-Hettinger	3212	Morrow Crossing	Kobe
2	Osinski and Sons	6495	Lyons Lane	Burunday
3	Gislason Group	7800	Westridge Alley	Allen
4	O'Reilly-Boehm	548	Porter Point	Labuan
5	Walter-Ritchie	7519	Portage Pass	DziaÅ□dowo
6	Luettgen, Dooley and Spinka	2980	Northland Pass	Yangkang
7	Blanda, Fisher and Harber	27938	Red Cloud Point	Adani
8	Lowe-Mante	30	Nancy Pass	Zheyuan
9	Bednar LLC	36	Magdeline Circle	VlkoÅ¡
10	Schinner-Volkman	90657	Menomonie Avenue	Middleton



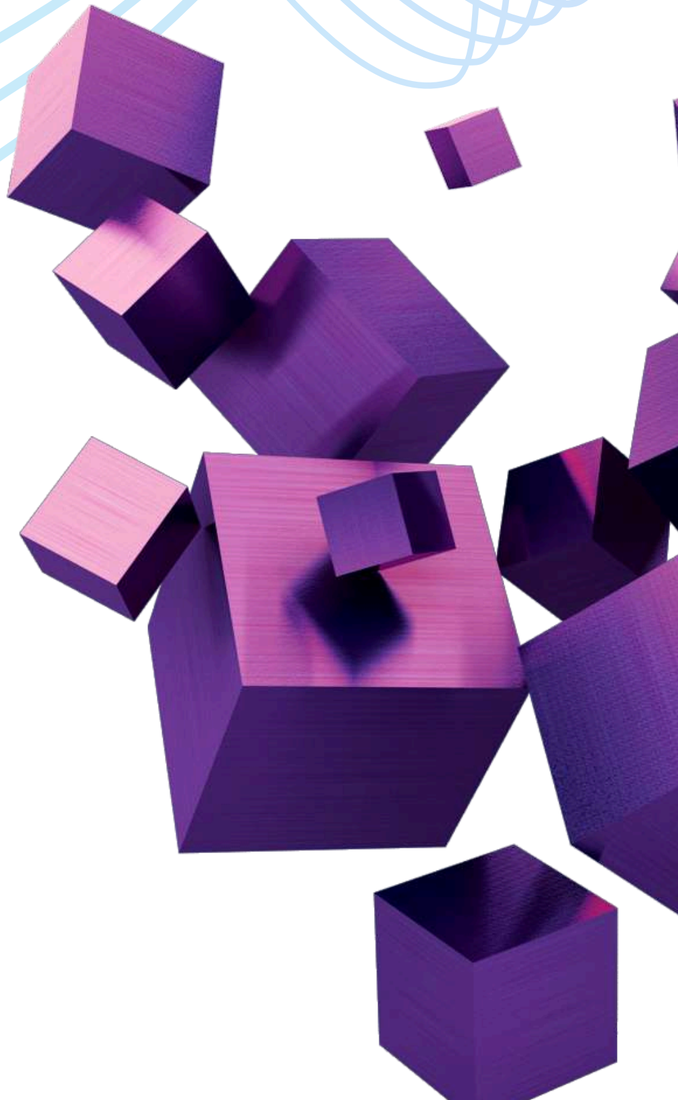
- Purpose: Retrieves all stores with their complete addresses
- Joins store and city tables to get city names
- Returns store ID, name, street number, street name, and city name

PRODUCT LIST WITH CATEGORIES

```sql
SELECT p.product_id, p.name AS product_name, pc.name AS category_name
FROM product_m p
JOIN product_category_m pc ON p.category_id = pc.category_id;
```

Results | Explain | Describe | Saved SQL | History

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NAME
1	Roasted Chickpeas	Food - Soups
2	Electric Hot Pot	Food - Dips
3	Beach Cover-Up	Toys
4	Honey Ginger Tea	Food - Snacks
5	Mediterranean Couscous Salad	Food - Condiments
6	Eco-Friendly Notepad	Food - Dips
7	Bicycle Lock	Kitchen
8	LED Disco Ball Light	Food - Condiments
9	Eco-Friendly Notepad	Food - Bakery
10	Basil Lemonade	Food - Soups



- Purpose: Lists all products with their categories
- Joins product and product_category tables
- Returns product ID, name, and category name

EMPLOYEE LIST WITH POSITIONS

```sql
SELECT e.employee_id, e.name, e.position
FROM EMPLOYEE_M e
```

Results | Explain | Describe | Saved SQL | History

EMPLOYEE_ID	NAME	POSITION
1	Croxton	Office Administrator
2	Charteris	Security Guard
3	Dallas	Stockroom Associate
4	Caddy	Department Supervisor
5	MacCostigan	Stock Clerk
6	Gurg	Office Administrator
7	Sange	Stockroom Associate
8	Ambrodi	Store Manager
9	Powles	Cashier
10	Wevell	Inventory Specialist

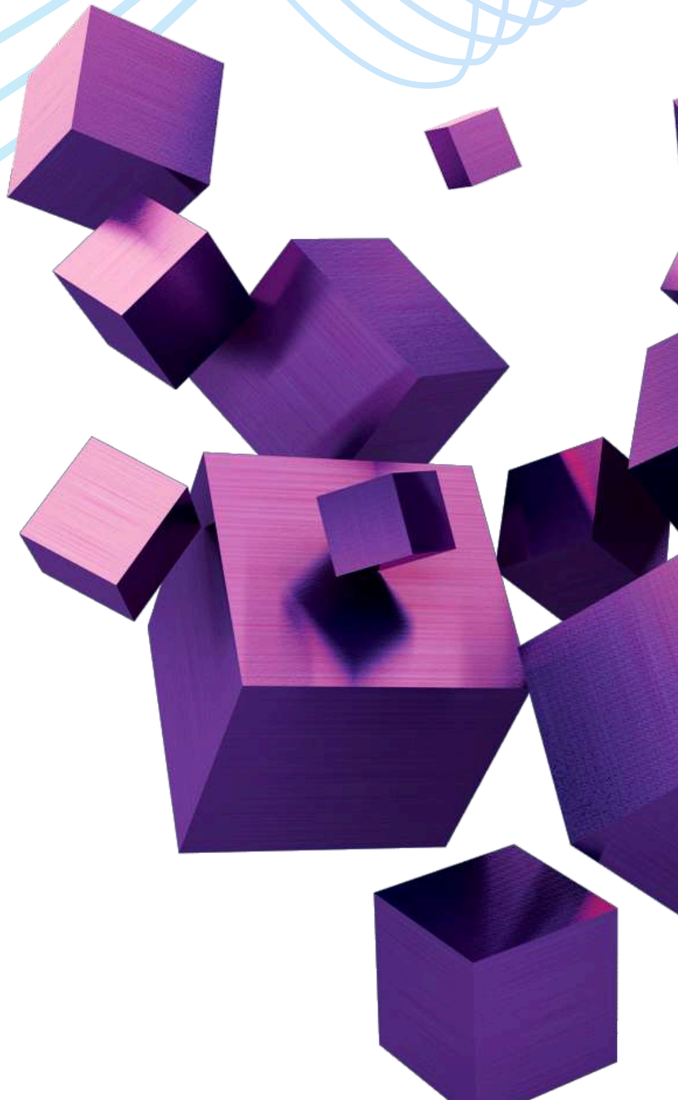
- Purpose: Retrieves basic employee information
- Simple SELECT from employee table
- Returns employee ID, first name, last name, and position

SUPPLIER LIST WITH CONTACT INFO

```sql
SELECT s.supplier_id, s.name, sc.phone, sc.email
 FROM supplier_m s
 JOIN supplier_contact_m sc ON s.supplier_id = sc.supplier_id
```

Results | Explain | Describe | Saved SQL | History

SUPPLIER_ID	NAME	PHONE	EMAIL
99	Topicloungе	277-886-4674	rgilder0@prnewswire.com
74	Thoughtstorm	634-480-1402	cmckernan1@sina.com.cn
77	Rhycero	679-977-8680	hdytham2@scientificamerican.com
15	Yodel	146-295-4867	gberling3@mashable.com
47	Edgetag	976-546-8837	mhalstead4@businesswire.com
59	Roombo	915-868-4591	tarchbould5@ifeng.com
100	Myworks	705-941-2388	cclemendet6@ox.ac.uk
31	Wordify	367-302-1059	lmerchant7@goodreads.com
18	Browsecat	863-209-8582	apoore8@dropbox.com
87	Buzzdog	941-139-7180	gspivey9@auda.org.au



- Purpose: Gets supplier information with contact details
- Joins supplier and supplier_contact tables
- Returns supplier ID, name, phone, and email

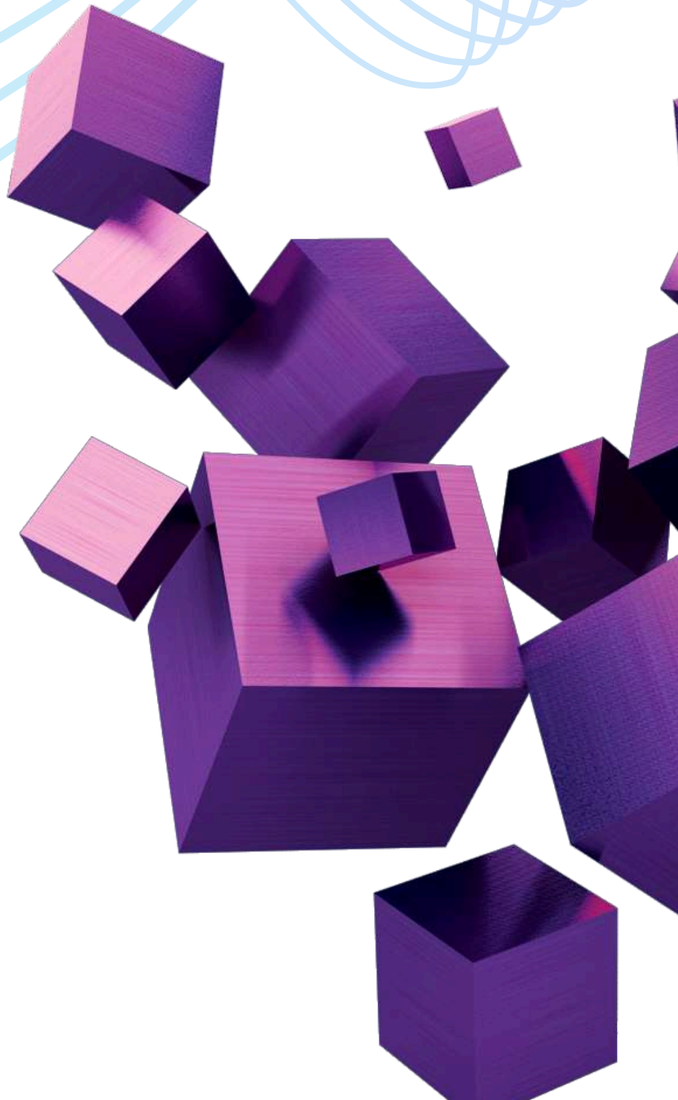
CUSTOMER PURCHASE COUNT

```sql
SELECT c.customer_id, c.name , COUNT(s.sale_id) AS total_purchases
 FROM customer_m c
 LEFT JOIN sale_m s ON c.customer_id = s.customer_id
 GROUP BY c.customer_id, c.name;
```

Results | Explain | Describe | Saved SQL | History

CUSTOMER_ID	NAME	TOTAL_PURCHASES
26	Teena	5
70	Egon	2
42	Bogey	2
14	Calhoun	1
88	Dorolisa	1
79	Marcus	1
86	Oswald	2
52	Angus	0
7	Cassie	0
63	Lizette	0

- Purpose: Counts total purchases per customer
- Uses LEFT JOIN to include customers with no purchases
- Groups results by customer an

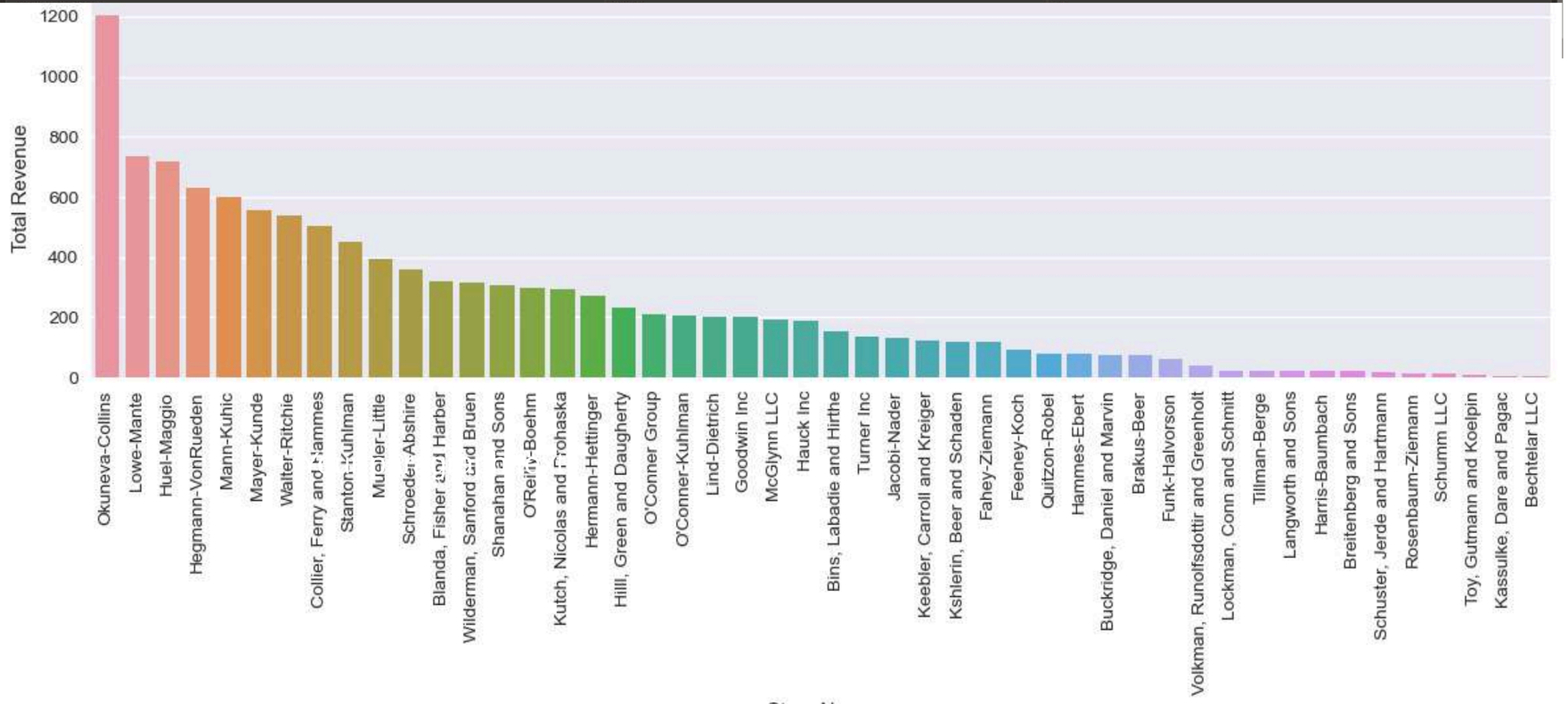


THE HIGHEST SALES REVENUE BY STORE

```sql
SELECT s.name AS store_name,
       SUM(si.quantity * si.price) AS total_revenue,
       COUNT(DISTINCT sa.sale_id) AS number_of_sales
FROM store_m s
JOIN sale_m sa ON s.store_id = sa.store_id
JOIN sale_item_m si ON sa.sale_id = si.sale_id
GROUP BY s.name
ORDER BY total_revenue DESC;
```

Results | Explain | Describe | Saved SQL | History

STORE_NAME	TOTAL_REVENUE	NUMBER_OF_SALES
Okuneva-Collins	1202.75	3
Lowe-Mante	735.62	3
Huel-Maggio	719.84	1
Hegmann-VonRueden	629.91	1
Mann-Kuhic	601.77	3
Mayer-Kunde	554.47	2
Walter-Ritchie	539.85	1
Collier, Ferry and Hammes	504.83	2



- Purpose: Analyzes sales performance
for each store and extract the highest revenue

- Joins store, sale, and sale_item tables to get complete sales information

- Groups data by store name

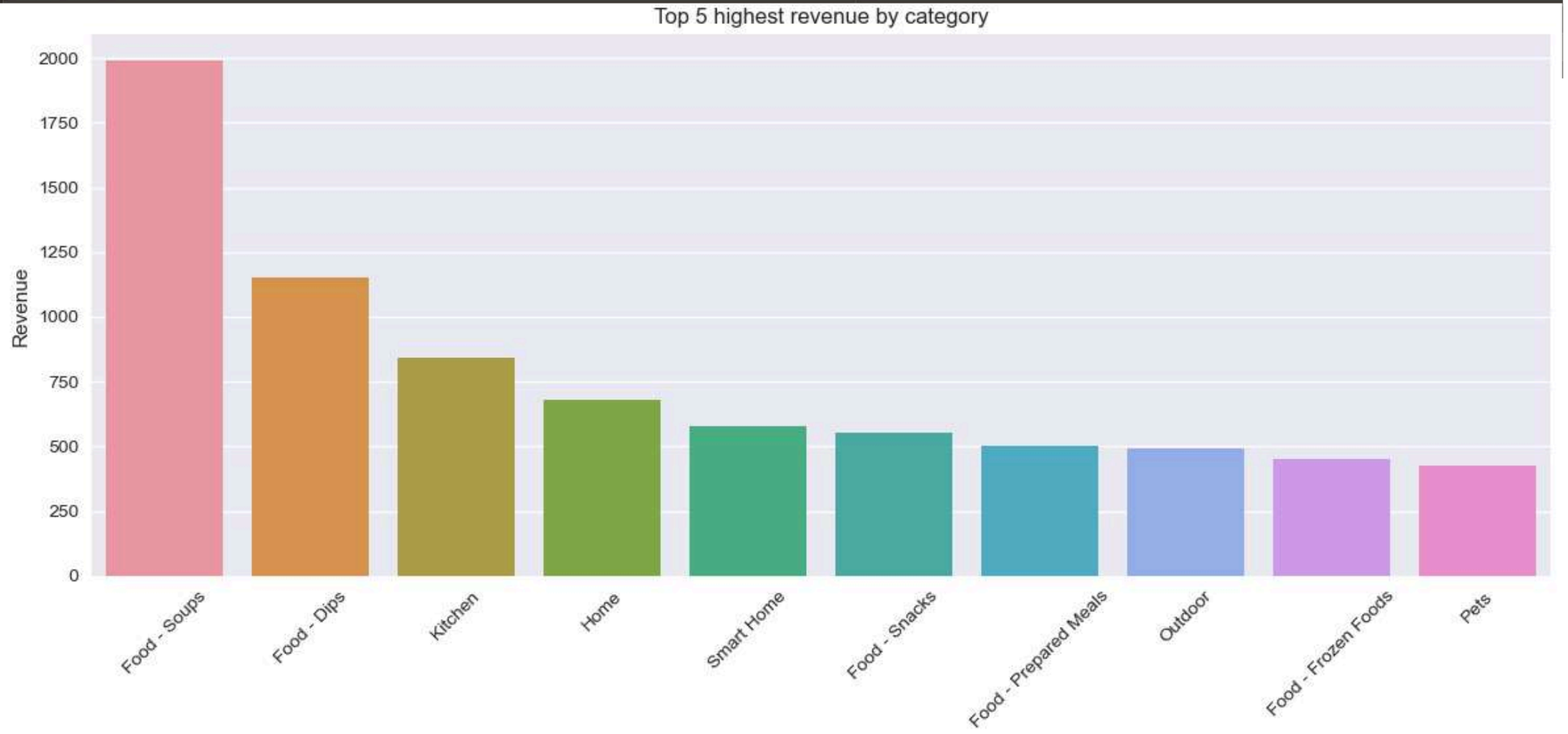
- Returns store name, month, total revenue, and number of sales transactions

PRODUCT PERFORMANCE ANALYSIS BY CATEGORY

```sql
SELECT
    pc.name AS category_name,
    SUM(si.quantity * si.price) AS total_revenue
FROM product_category_m pc
JOIN product_m p ON pc.category_id = p.category_id
JOIN sale_item_m si ON p.product_id = si.product_id
GROUP BY pc.name
ORDER BY total_revenue DESC
```

Results | Explain | Describe | Saved SQL | History

CATEGORY_NAME	TOTAL_REVENUE
Food - Soups	1990.31
Food - Dips	1150.64
Kitchen	843.52
Home	680.29
Smart Home	576.32
Food - Snacks	551.51
Food - Prepared Meals	500.77



- Purpose: Evaluates product category highest revenue

- Joins product_category, product, and sale_item tables

- Groups data by product category

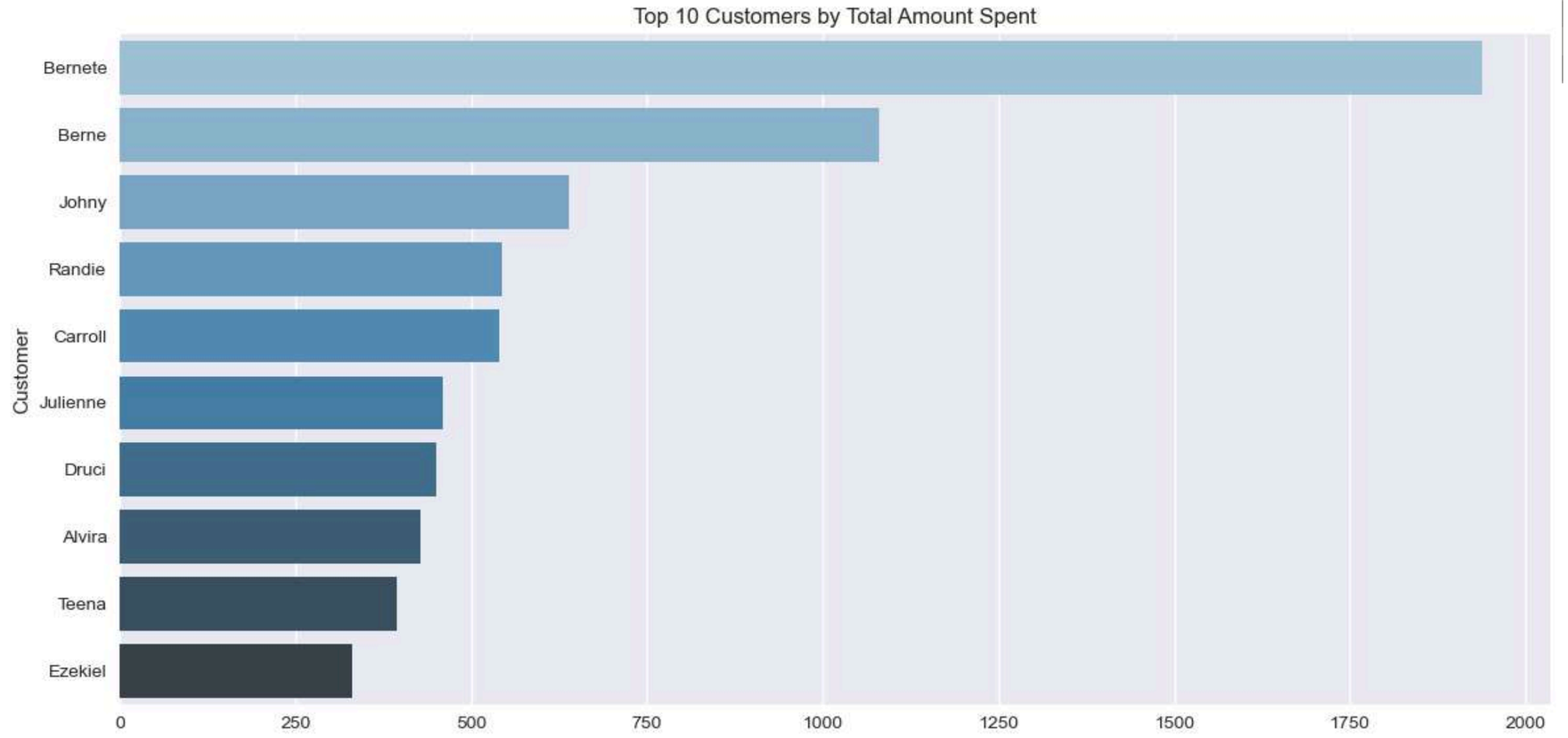
- Returns category statistics including number of products, total sales

CUSTOMER PURCHASE BEHAVIOR ANALYSIS

```sql
SELECT
    c.name AS customer_name,
    COUNT(s.sale_id) AS purchases_count,
    SUM(si.quantity * si.price) AS total_spent
FROM customer_m c
JOIN sale_m s ON c.customer_id = s.customer_id
JOIN sale_item_m si ON s.sale_id = si.sale_id
GROUP BY c.name
ORDER BY total_spent DESC;
```

Results | Explain | Describe | Saved SQL | History

CUSTOMER_NAME	PURCHASES_COUNT	TOTAL_SPENT
Bernete	3	1938.74
Berne	4	1079.75
Johny	6	638.27
Randie	3	544.48
Carroll	2	539.85



- Purpose: Analyzes customer purchasing patterns

- Joins customer, sale, and sale_item tables

- Groups data by customer

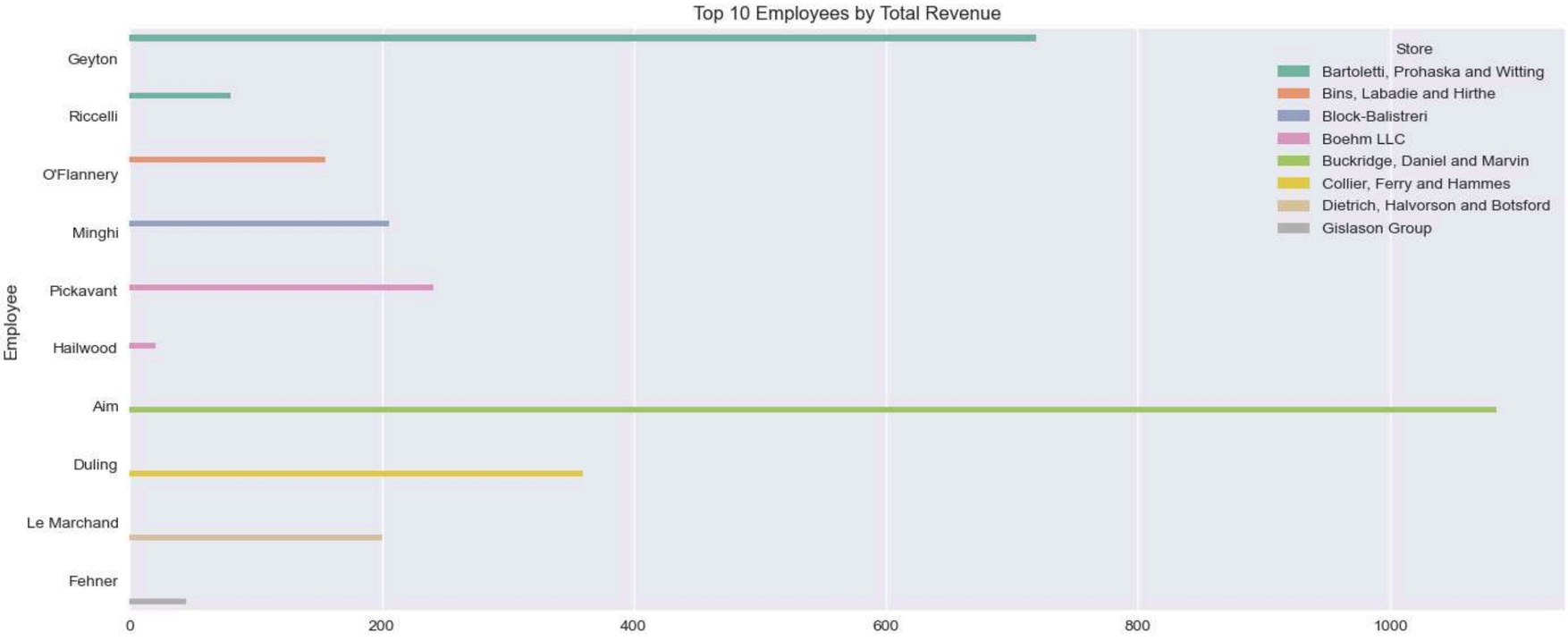
- Returns customer purchase statistics including frequency, total spending, and average purchase value

EMPLOYEE PERFORMANCE BY STORE

```sql
SELECT
    s.name AS store_name,
    e.name AS employee_name,
    COUNT(DISTINCT sa.sale_id) AS number_of_sales,
    SUM(si.quantity * si.price) AS total_revenue,
    AVG(si.quantity * si.price) AS average_sale_value
FROM employee_m e
JOIN store_m s ON e.store_id = s.store_id
JOIN sale_m sa ON e.employee_id = sa.employee_id
JOIN sale_item_m si ON sa.sale_id = si.sale_id
GROUP BY s.name, e.employee_id, e.name
ORDER BY s.name, total_revenue DESC;
```

Results | Explain | Describe | Saved SQL | History

STORE_NAME	EMPLOYEE_NAME	NUMBER_OF_SALES	TOTAL_REVENUE	AVERAGE_SALE_VALUE
Bartoletti, Prohaska and Witting	Geyton	3	719.03	239.6766666666666666666666666666666667
Bartoletti, Prohaska and Witting	Riccelli	1	79.97	39.985
Bins, Labadie and Hirthe	O'Flannery	1	155.34	51.78
Block-Balistreri	Minghi	1	205.87	102.935
Boehm LLC	Pickavant	1	240.89	120.445
Boehm LLC	Hailwood	1	20.95	20.95
Buckridge, Daniel and Marvin	Aim	2	1084.33	216.866



- Purpose: Evaluates employee sales performance

- Joins employee, store, sale, and sale_item tables

- Groups data by store and employee

- Returns sales statistics for each employee including number of sales and revenue generated

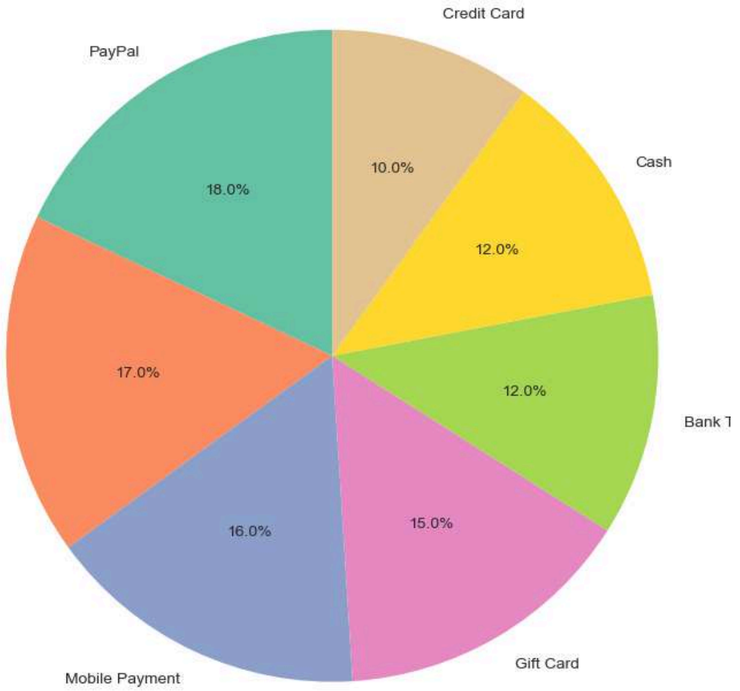
PAYMENT METHOD ANALYSIS

```sql
SELECT p.METHOD,
       COUNT(p.PAYMENT_ID) AS number_of_payments,
       SUM(p.AMOUNT) AS total_amount,
       AVG(p.AMOUNT) AS average_payment,
       COUNT(p.PAYMENT_ID) * 100.0 / (SELECT COUNT(*) FROM PAYMENT_M) AS percentage_of_total
FROM PAYMENT_M p
GROUP BY p.METHOD
ORDER BY number_of_payments DESC;
```

Results | Explain | Describe | Saved SQL | History

METHOD	NUMBER_OF_PAYMENTS	TOTAL_AMOUNT	AVERAGE_PAYMENT	PERCENTAGE_OF_
PayPal	18	11724116.58	651339.81	18
Debit Card	17	9470564.7	557092.0411764705882352941176470588235 29	17
Mobile Payment	16	9926085.14	620380.32125	16
Gift Card	15	7830421.79	522028.1193333333333333333333333333333	15
Bank Transfer	12	5319404.53	443283.7108333333333333333333333333333333	12
Cash	12	5865862.7	488821.8916666666666666666666666666666667	12
Credit Card	10	6316315.87	631631.587	10

Percentage of Total Payments by Payment Method



- Purpose: Analyzes payment method preferences

- Uses only the payment table

- Groups data by payment method

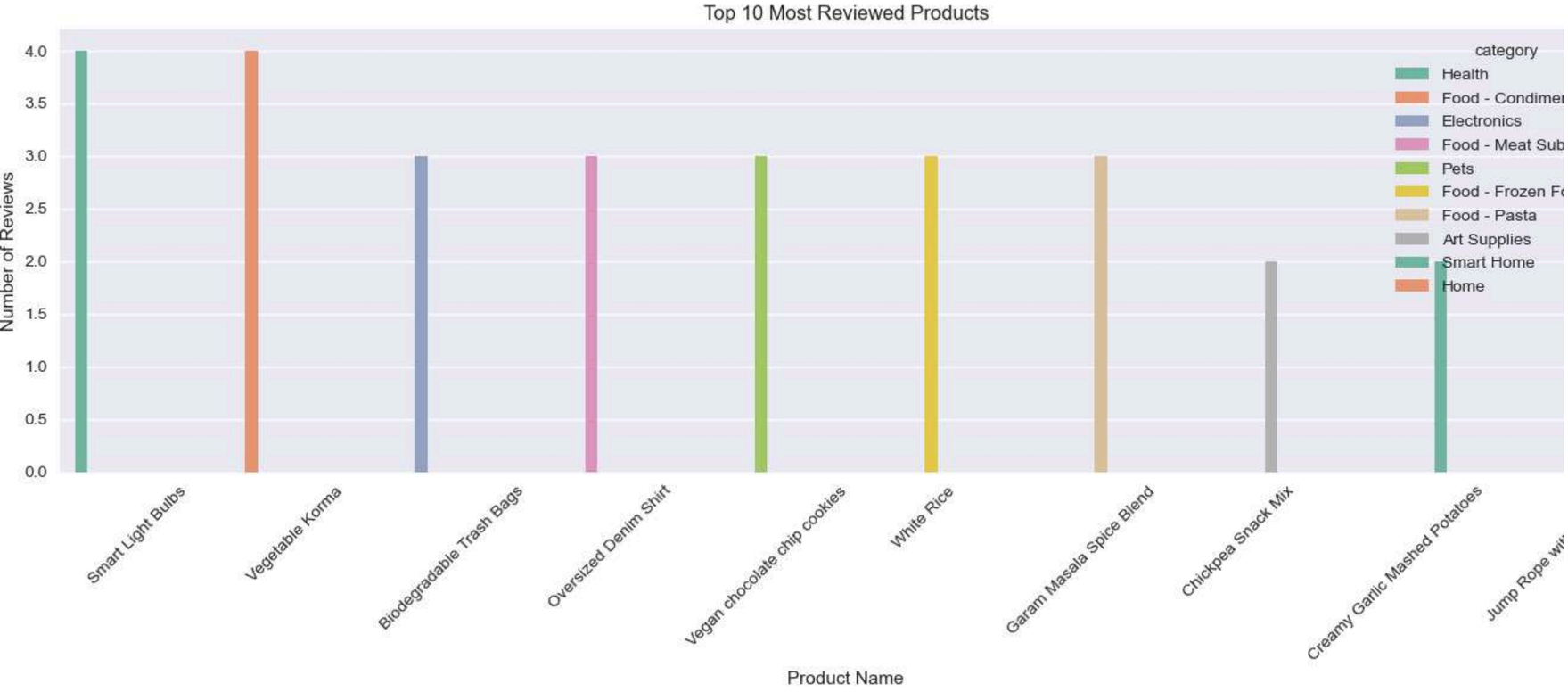
- Returns payment statistics including volume, value, and percentage distribution

CUSTOMER FEEDBACK ANALYSIS BY PRODUCT

```sql
SELECT p.name AS product_name,
       pc.name AS category,
       COUNT(f.feedback_id) AS number_of_reviews,
       AVG(f.rating) AS average_rating
FROM product_m p
JOIN product_category_m pc ON p.category_id = pc.category_id
JOIN feedback_m f ON p.product_id = f.product_id
GROUP BY p.name, pc.name
ORDER BY number_of_reviews DESC, average_rating DESC;
```

Results | Explain | Describe | Saved SQL | History

PRODUCT_NAME	CATEGORY	NUMBER_OF_REVIEWS	AVERAGE_RATING
Smart Light Bulbs	Health	4	4.25
Vegetable Korma	Food - Condiments	4	3.25
Biodegradable Trash Bags	Electronics	3	3.3333333333333333333333333333333333333
Oversized Denim Shirt	Food - Meat Substitutes	3	2.6666666666666666666666666666666666667
White Rice	Food - Frozen Foods	3	2.6666666666666666666666666666666666667
Vegan chocolate chip cookies	Pets	3	2.6666666666666666666666666666666666667
Garam Masala Spice Blend	Food - Pasta	3	2



- Purpose: Analyzes customer feedback for products

- Joins product, product_category, and feedback tables

- Groups data by product

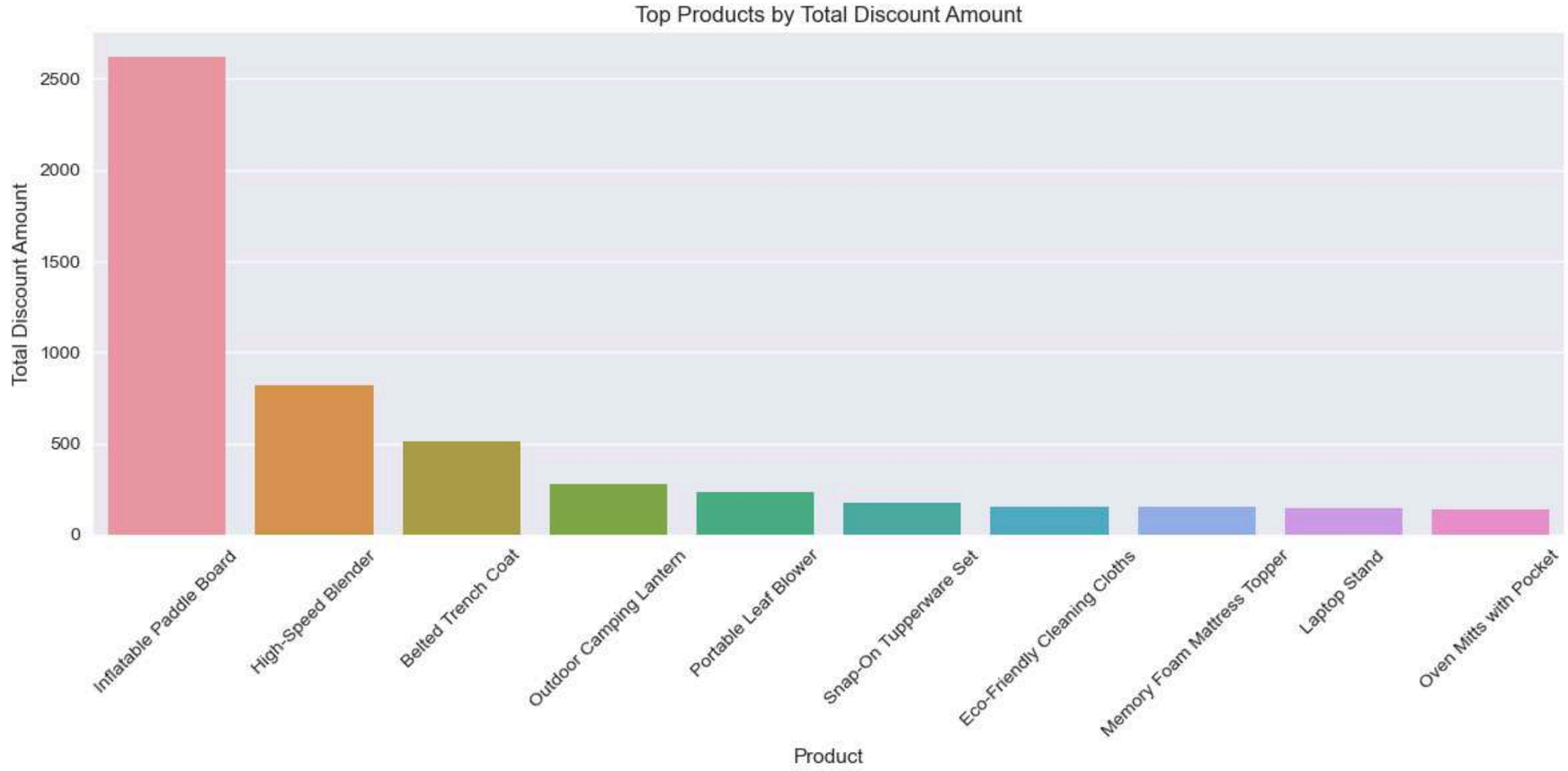
- Returns review statistics including average rating and percentage of positive reviews

DISCOUNT IMPACT ANALYSIS

```sql
SELECT
    p.name AS product_name,
    d.PERSENTAGE AS discount_percentage,
    COUNT(si.sale_id) AS number_of_discounted_sales,
    SUM(si.quantity * si.price * d.PERSENTAGE / 100) AS total_discount_amount
FROM DISCOUNT_M d
JOIN product_m p ON d.product_id = p.product_id
JOIN sale_item_m si ON p.product_id = si.product_id
JOIN sale_m s ON si.sale_id = s.sale_id
GROUP BY p.name, d.PERSENTAGE
ORDER BY total_discount_amount DESC;
```

Results | Explain | Describe | Saved SQL | History

PRODUCT_NAME	DISCOUNT_PERCENTAGE	NUMBER_OF_DISCOUNTED_SALES	TOTAL_DISCOUNT_AMOUNT
Roasted Chickpeas	67	3	849.0843
Oven Mitts with Pocket	88	1	563.1296
Outdoor Camping Lantern	87	1	548.0217
Creamy Garlic Mashed Potatoes	86	2	436.3382



- Purpose: Analyzes customer feedback for products

- Joins product, product_category, and feedback tables

- Groups data by product

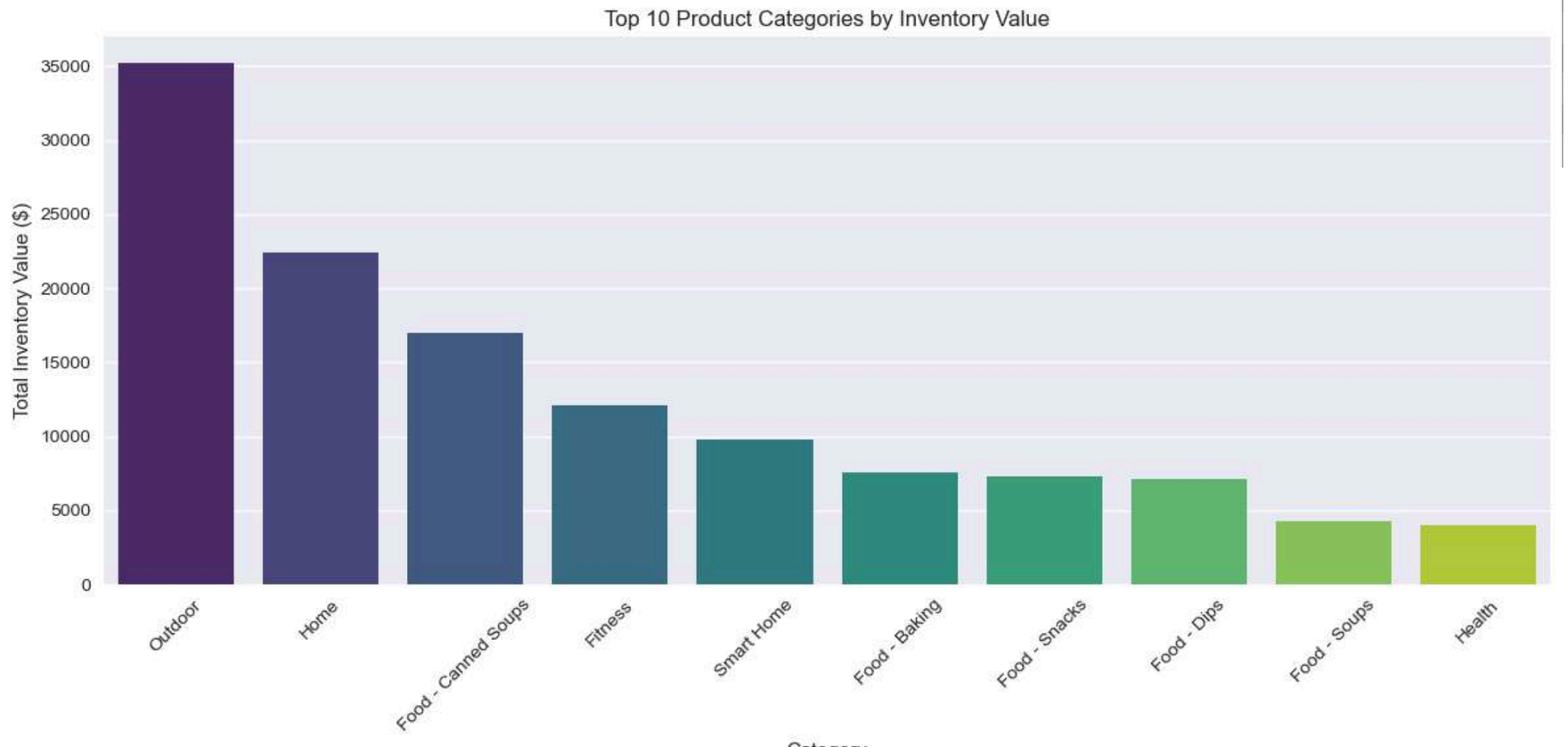
- Returns review statistics including average rating and percentage of positive reviews

INVENTORY TURNOVER ANALYSIS

```sql
SELECT
    pc.name AS category_name,
    COUNT(p.product_id) AS product_count,
    SUM(i.quantity) AS total_inventory,
    SUM(i.quantity * p.price) AS total_inventory_value,
    AVG(p.price) AS average_product_price
FROM
    product_m p
JOIN
    product_category_m pc ON p.category_id = pc.category_id
JOIN
    inventory_m i ON p.product_id = i.product_id
GROUP BY
    pc.name
ORDER BY
    total_inventory_value DESC
```

Results | Explain | Describe | Saved SQL | History

CATEGORY_NAME	PRODUCT_COUNT	TOTAL_INVENTORY	TOTAL_INVENTORY_VALUE	AVERAGE_PRODUCT_PRICE
Outdoor	8	415	35193.85	107.99
Home	9	429	22381.71	45.1011111111111111111111111111111111
Food - Canned Soups	2	68	16999.32	249.99



-- Purpose: Analyzes product performance by category

-- Joins product, productCategory, and inventory tables

-- Groups data by product category

-- Calculates total inventory value and average price per category

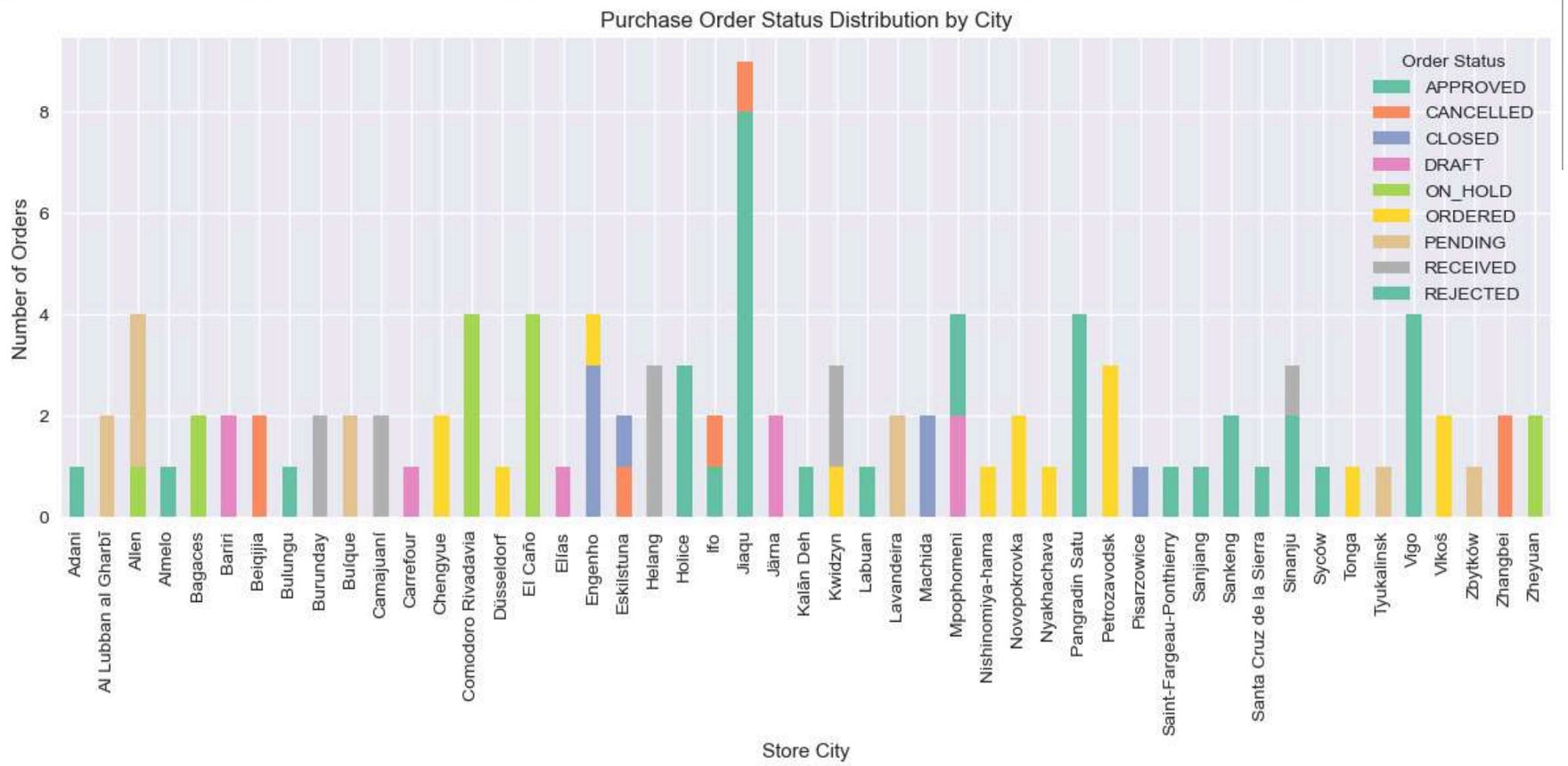
-- Helps identify most valuable inventory categories

ANALYZES PURCHASE ORDER PATTERNS BY SUPPLIER AND STORE LOCATION

```sql
SELECT
    p.Supplier AS supplier_name,
    c.name AS store_city,
    COUNT(po.purchase_order_id) AS total_orders,
    SUM(CASE WHEN po.status = 'RECEIVED' THEN 1 ELSE 0 END) AS received_orders,
    SUM(CASE WHEN po.status = 'REJECTED' THEN 1 ELSE 0 END) AS rejected_orders,
    ROUND(100.0 * SUM(CASE WHEN po.status = 'RECEIVED' THEN 1 ELSE 0 END) /
        COUNT(po.purchase_order_id), 1) AS fulfillment_rate
FROM
    purchase_order_m po
JOIN
    product_m p ON po.supplier_id = p.Supplier  -- Matching supplier IDs
JOIN
    city_m c ON po.store_id = c.city_id  -- Matching store locations
GROUP BY
    p.Supplier, c.name
ORDER BY
    total_orders DESC, fulfillment_rate DESC
```

Results | Explain | Describe | Saved SQL | History

SUPPLIER_NAME	STORE_CITY	TOTAL_ORDERS	RECEIVED_ORDERS	REJECTED_ORDERS	FULFILLMENT_RATE
73	Jiaqu	6	0	0	0
26	Pangradin Satu	4	0	0	0



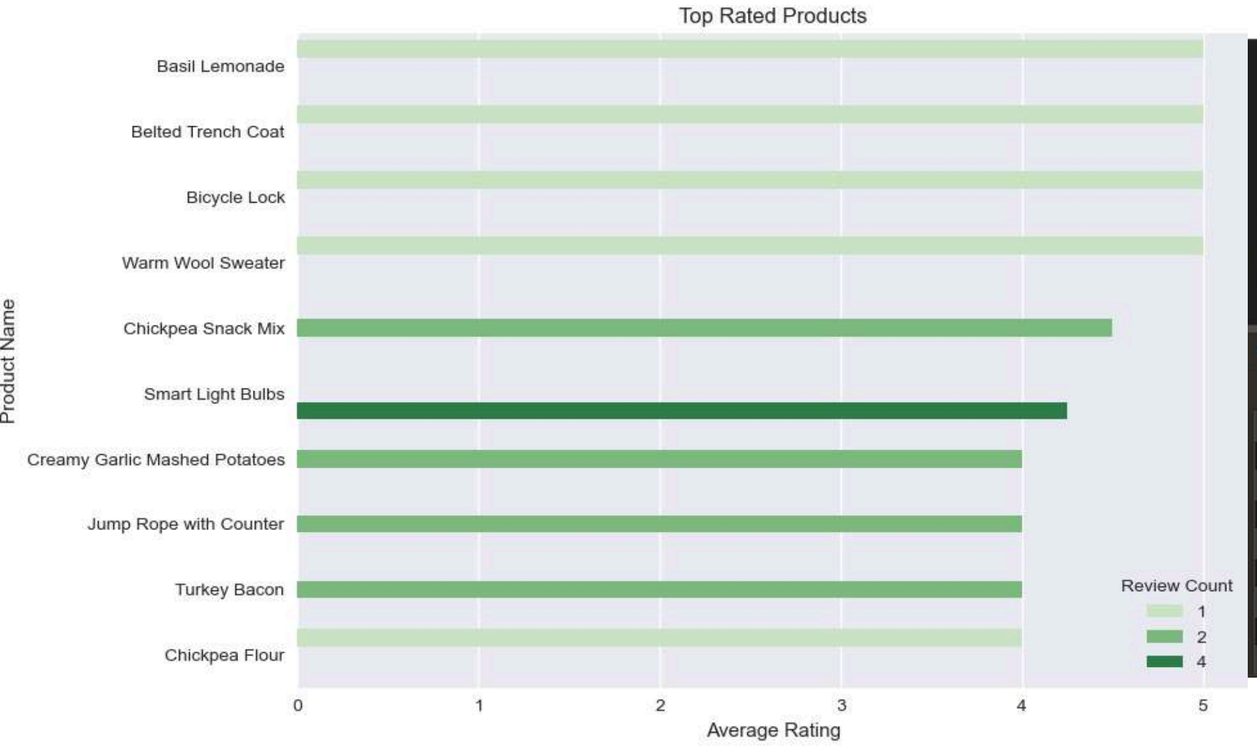
-- Purpose: Analyzes purchase order patterns by supplier and store location

-- Joins purchaseOrder, product_sql (for supplier), and city (for store location) tables

-- Calculates order volume and status distribution by supplier and location

-- Helps identify reliable suppliers and geographic purchasing patterns

PRODUCTS WITH HIGH CUSTOMER RATINGS




```sql
SELECT
    p.name AS product_name,
    COUNT(f.feedback_id) AS review_count,
    ROUND(AVG(f.rating), 2) AS average_rating
FROM
    feedback_m f
JOIN
    product_m p ON f.product_id = p.product_id
WHERE
    f.rating >= 1
GROUP BY
    p.name
HAVING
    COUNT(f.feedback_id) >= 1
ORDER BY
    review_count DESC
```

Results | Explain | Describe | Saved SQL | History

PRODUCT_NAME	REVIEW_COUNT	
Smart Light Bulbs	4	4.25
Vegetable Korma	4	3.25
White Rice	3	2.67
Garam Masala Spice Blend	3	2
Eco-Friendly Notepad	3	3
Biodegradable Trash Bags	3	3.33
Vegan chocolate chip cookies	3	2.67
Oversized Denim Shirt	3	2.67
Chickpea Snack Mix	2	4.5

-- PURPOSE: FINDS PRODUCTS WITH HIGH CUSTOMER RATINGS (4-5 STARS) THAT HAVE AT LEA ST 5 REVIEWS

-- JOINS FEEDBACK AND PRODUCT TABLES

-- FILTERS FOR HIGHLY RATED PRODUCTS WITH SIGNIFICANT REVIEW VOLUME

-- SIMPLE BUT USEFUL FOR IDENTIFYING BEST-SELLING QUALITY PRODUCTS

Subqueries

```sql
SELECT s.store_id, s.name, SUM(si.quantity * si.price) AS total_sales
FROM store s
JOIN sale sa ON s.store_id = sa.store_id
JOIN saleItem si ON sa.sale_id = si.sale_id
GROUP BY s.store_id, s.name
HAVING SUM(si.quantity * si.price) > (
    SELECT AVG(store_sales)
    FROM (
        SELECT SUM(si.quantity * si.price) AS store_sales
        FROM saleItem si
        JOIN sale sa ON si.sale_id = sa.sale_id
        GROUP BY sa.store_id
    )
);
```

This query displays stores where total sales exceed the average for all stores.
The inner subquery calculates the sales for each store and finds the average.
The outer query for sales is calculated for each store and selects only those where the minimum is higher than the average.
This allows you to think about which stores are doing better than others.

STORE_ID	NAME	TOTAL_SALES
7	Blanda, Fisher and Harber	321.22
71	Okuneva-Collins	1202.75
5	Walter-Ritchie	539.85
35	Stanton-Kuhlman	449.9
85	Mayer-Kunde	554.47
8	Lowe-Mante	735.62
21	Wilderman, Sanford and Bruen	313.82
45	Mueller-Little	394.23
27	Hegmann-VonRueden	629.91
54	Kutch, Nicolas and Prohaska	294.89

```sql
SELECT s.supplier_id, s.name,
       (SELECT COUNT(*)
        FROM purchaseOrder po
        WHERE po.supplier_id = s.supplier_id) AS total_orders
FROM Supplier s
ORDER BY total_orders DESC;
```

This query shows the suppliers with the most orders.
For each supplier, it counts how many orders (from the purchaseOrder table) are associated with it.
The results are sorted by the number of orders in descending order.
This is important for identifying the most active or in-demand suppliers.

SUPPLIER_ID	NAME	TOTAL_ORDERS
39	Realbridge	5
44	Livepath	4
70	Meevee	3
80	Demizz	3
42	Rhybox	3
21	Cogidoo	3
26	Blogtags	3
82	Ntags	3
27	Brainverse	2
71	Twimbo	2

More than 10 rows available. Increase rows selector to view more rows.

```sql
SELECT c.customer_id, c.name,
       (SELECT AVG(total)
        FROM (
            SELECT SUM(si.quantity * si.price) AS total
            FROM sale s
            JOIN saleItem si ON s.sale_id = si.sale_id
            WHERE s.customer_id = c.customer_id
            GROUP BY s.sale_id
        )
       ) AS avg_order_value
FROM customer c;
```

This query calculates the average order amount for each customer. The nested subquery first calculates the total amount for each customer order, then calculates the average of these amounts. Based on the data, it is clear that only 30% of customers (3 out of 10) made purchases. This is reflected in the presence of average check values. The remaining customers are registered, but are not yet active in the sales system.

CUSTOMER_ID	NAME	AVG_ORDER_VALUE
1	Jeffy	240.89
2	Paxton	-
3	Isabel	-
4	Ingram	-
5	Hunfredo	-
6	Delmore	-
7	Cassie	-
8	Betty	-
9	Indira	44.99
10	Randie	544.48

More than 10 rows available. Increase rows selector to view more rows.

10 rows returned in 0.00 seconds Download

```sql
SELECT c.customer_id, c.name,
       (SELECT COUNT(DISTINCT s.store_id)
        FROM sale s
        WHERE s.customer_id = c.customer_id) AS stores_visited
FROM customer c
WHERE (SELECT COUNT(DISTINCT s.store_id)
       FROM sale s
       WHERE s.customer_id = c.customer_id) > 1;
```

This subquery finds customers who have shopped at more than one store. This may indicate loyalty to a store chain or diversity of customer needs. This analytics is useful for marketing and customer program.

CUSTOMER_ID	NAME	STORES_VISITED
1	Jeffy	2
12	Cesare	2
17	June	2
21	Humbert	3
22	Gina	3
23	Farah	2
24	Ezekiel	3
26	Teena	4
27	Bernete	3
28	Maurise	2

```sql
SELECT p.Name, AVG(f.rating) AS avg_rating
FROM feedback f
JOIN product p ON f.product_ID = p.product_ID
WHERE f.rating > (
    SELECT AVG(rating)
    FROM feedback
)
GROUP BY p.Name
ORDER BY avg_rating DESC;
```

This query identifies products that have received reviews with scores higher than the average across all reviews. It combines the product and review tables, calculates the average score only among positive reviews, and sorts the products by average score descending. This analysis helps identify products with the highest customer satisfaction.

NAME	AVG_RATING
Fruit and Nut Energy Bites	5
Honey BBQ Riblets	5
Basil Lemonade	5
Belted Trench Coat	5
Warm Wool Sweater	5
Bicycle Lock	5
Chickpea Snack Mix	4.5
Smart Light Bulbs	4.25
Biodegradable Trash Bags	4
Dark Chocolate Covered Pretzels	4

More than 10 rows available. Increase rows selector to view more rows.

```sql
SELECT name
FROM Produc
WHERE product_id IN (
    SELECT product_id
    FROM Discount
    WHERE CURRENT_DATE BETWEEN start_date AND end_date
);
```

This query displays products that have active discounts.
The inner subquery selects the product IDs of all discounts that are currently active by comparing the current date to the discount's start and end dates.
The outer query then retrieves the names of the products whose IDs were returned by the subquery.
This helps to identify which products are currently on sale.

NAME
Roasted Chickpeas
Basil Lemonade
Water Bottle
Jump Rope with Counter
Caramelized Onion Dip
Biodegradable Trash Bags
Avocado Oil Mayo

```sql
SELECT name
FROM Customer
WHERE customer_id IN (
    SELECT customer_id
    FROM Salee
    WHERE sale_id IN (
        SELECT sale_id
        FROM SaleItem
        WHERE sale_item_id IN (
            SELECT sale_item_id
            FROM Return
        )
    )
);
```

This query identifies customers who have returned at least one product.
The innermost subquery selects the sale item IDs from the Return table.
The next subquery up selects the sale IDs for those sale items
The next subquery selects the customer IDs for those sales.
Finally, the outer query retrieves the names of those customers.
This allows for focusing on customers with return behavior.

NAME
Teena
Randie
Bernete
Bogey
Vanya
Jeffy
Hyacinthia
Gina
Falkner

```sql
SELECT name, price
FROM Produc
WHERE (category_id, price) IN (
    SELECT category_id, MAX(price)
    FROM Produc
    GROUP BY category_id
);
```

This query finds the most expensive product within each product category.
The inner subquery determines the maximum price for each category by grouping the products by their category ID and using the MAX() function.
The outer query then selects the name and price of those products that match the category and the maximum price for that category found in the subquery.
This is useful for analyzing the highest-priced items in different categories.

NAME	PRICE
SSD Drive	120
Beach Cover-Up	29.99
Honey Ginger Tea	2.99
Bicycle Lock	29.99
LED Disco Ball Light	19.99
Eco-Friendly Notepad	8.99
Basil Lemonade	2.99
Water Bottle	18.99
Jump Rope with Counter	12.99
Samsung Galaxy Smartwatch	249.99

```sql
SELECT name,position
FROM Employee
WHERE employee_id NOT IN (
    SELECT DISTINCT employee_id
    FROM Salee
);
```

This query finds employees who have not made any sales.
The inner subquery selects a distinct list of all employee IDs that are present in the Sale table, indicating they have handled sales.
The outer query then retrieves the names of employees whose IDs are NOT in that list.
This helps in identifying employees who may need training or have different roles.

NAME	POSITION
Charteris	Security Guard
Caddy	Department Supervisor
MacCostigan	Stock Clerk
Villar	Greeter
Beange	Assistant Manager
Guys	Janitor
MacInnes	Department Supervisor
Bengough	Customer Service Representative
Higginbottam	Department Supervisor
Greer	Security Guard

```sql
SELECT name
FROM Produc
WHERE product_id NOT IN (
    SELECT product_id
    FROM SaleItem
);
```

This query lists products that have never been sold.
The inner subquery selects the product IDs of all products that appear in the SaleItem table, meaning they have been sold at least once.
The outer query retrieves the names of products whose IDs are NOT in that list.
This can help in inventory management and identifying unpopular products.

NAME
SSD Drive
Electric Hot Pot
Beach Cover-Up
Mediterranean Couscous Salad
LED Disco Ball Light
Jump Rope with Counter
Caramelized Onion Dip
Oversized Denim Shirt
LED Desk Lamp with USB Charging
Lentil Soup (canned)

```oracle_plsql
CREATE OR REPLACE TRIGGER trg_set_return_date
BEFORE INSERT ON Return
FOR EACH ROW
BEGIN
    IF :NEW.return_date IS NULL THEN
        :NEW.return_date := SYSDATE;
    END IF;
END;
```

This trigger automatically sets the return date when a new return record is inserted.
It is a BEFORE INSERT trigger on the Return table, meaning it executes before a new row is added.
Inside the trigger, it checks if the 'return_date' for the new row is NULL.
If it is NULL, it sets the 'return_date' to the current system date (SYSDATE).
This ensures that every return record has a date, even if it's not manually provided.

```oracle_plsql
INSERT INTO Return (return_id, sale_item_id, quantity, reason)
VALUES (110, 1, 1, 'Defective product');
```

```oracle_plsql
-- INSERT INTO Return (return_id, sale_item_id, quantity, reason)
-- VALUES (110, 1, 1, 'Defective product');
select * from return where return_id = 110
```

Results | Explain | Describe | Saved SQL | History

ID	RETURN_ID	RETURN_DATE	QUANTITY	SALE_ITEM_ID	REASON
121	110	4/22/2025	1	1	Defective product

1 rows returned in 0.03 seconds Download

```sql
CREATE OR REPLACE TRIGGER trg_deletediscountson_product_delete
AFTER DELETE ON Produc
FOR EACH ROW
BEGIN
  DELETE FROM Discount
  WHERE product_id = :OLD.product_id;
END;
```

```sql
DELETE FROM Produc WHERE product_id = 11;
```

```sql
SELECT * FROM Discount WHERE product_id = 11;
```

Results | Explain | Describe | Saved SQL | History

no data found

This trigger deletes discounts associated with a product when that product is deleted.
This ensures that discounts are cleaned up when a product is removed from the system.

```sql
CREATE OR REPLACE TRIGGER trg_cascade_delete_inventory
AFTER DELETE ON Produc
FOR EACH ROW
BEGIN
    DELETE FROM Inventory
    WHERE product_id = :OLD.product_id;
END;
```

This trigger automatically deletes inventory records when a product is deleted.
It is an AFTER DELETE trigger on the Product table, running after a product is removed.
It deletes all records from the Inventory table where the 'product_id' matches the 'product_id' of the deleted product.
This prevents orphan records in the Inventory table and keeps the database consistent.

```sql
DELETE FROM Produc WHERE product_id = 7777;
```

Results | Explain | Describe | Saved SQL | History

1 row(s) deleted.

0.05 seconds

```sql
-- Add a product
-- INSERT INTO Produc (product_id, name, price, category_id, supplier)
-- VALUES (7777, 'Cascade Test Product', 300, 1, 1);
-- Add inventory for the product
-- INSERT INTO Inventory (inventory_id, product_id, store_id, quantity)
-- VALUES (9001, 7777, 1, 50);
-- Delete the product — trigger will delete related inventory automatically
-- DELETE FROM Produc WHERE product_id = 7777;
-- Check if inventory is gone
SELECT * FROM Inventory WHERE product_id = 7777;
```

Results | Explain | Describe | Saved SQL | History

no data found

This trigger prevents the inventory quantity from becoming negative.
It is a BEFORE INSERT OR UPDATE trigger on the Inventory table, executing before any changes to the quantity.
It checks if the new 'quantity' value is less than 0.
If it is, the trigger raises an application error (-20002) with a message indicating that the quantity cannot be negative.
This maintains data integrity by ensuring valid inventory levels.

```oracle-plsql
CREATE OR REPLACE TRIGGER trg_inventory_non_negative
BEFORE INSERT OR UPDATE ON Inventory
FOR EACH ROW
BEGIN
    IF :NEW.quantity < 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Quantity cannot be negative.');
    END IF;
END;
```

```oracle-plsql
INSERT INTO Inventory (inventory_id, product_id, store_id, quantity)
VALUES (101, 1, 1, -10);
```

esults | Explain | Describe | Saved SQL | History

```
ORA-20002: Quantity cannot be negative.
ORA-06512: at "WKSP_ZHANNURSQL.TRG_INVENTORY_NON_NEGATIVE", line 3
ORA-04088: error during execution of trigger
'WKSP_ZHANNURSQL.TRG_INVENTORY_NON_NEGATIVE'
```

```oracle_plsql
CREATE OR REPLACE TRIGGER update_inventory_after_sale
AFTER INSERT ON SaleItem
FOR EACH ROW
DECLARE
    PRAGMA autonomous_transaction;
BEGIN
    UPDATE Inventory
    SET quantity = quantity - :NEW.quantity
    WHERE product_id = :NEW.product_id;
    COMMIT;
END;
```

This trigger updates the inventory quantity after a sale is recorded.
It is an AFTER INSERT trigger on the SaleItem table, so it runs after a new sale item is added.
It declares a PRAGMA AUTONOMOUS_TRANSACTION, which allows the trigger to perform database operations independently of the main transaction.
The trigger updates the Inventory table, decreasing the 'quantity' of the sold product by the 'quantity' specified in the SaleItem record.
It uses COMMIT to save the inventory changes.
This keeps the inventory levels accurate as sales occur.

Conclusion

In this project, we designed and implemented a fully functional Retail Store & Inventory Management System. The system includes a well-structured database with 17 normalized tables, over 30 analytical SQL queries, and 5+ powerful triggers that automate key processes like inventory updates, product returns, and data cleanup.

We successfully:

Built an Entity-Relationship model with real-life connections.

Populated the database with realistic test data.

Created complex queries to analyze customer behavior, product performance, and store revenue.

Ensured data integrity and automation using SQL triggers.

Considered performance and scalability by applying normalization and indexing.

This project helped us better understand real-world database design, team collaboration, and the importance of analytics in business decision-making.