RETAIL DATA ANALYSIS USING SQL

🔷 Overview:

This project involves analyzing a retail company's operations using SQL. The dataset contains 10 related tables that represent core areas of a retail business such as products, orders, staff, customers, inventory, and stores.

By writing SQL queries, we aim to answer real-world business questions related to sales performance, inventory availability, staff productivity, and order processing.

🔷 Tools & Skills Used:

- **Google BigQuery –** to store and analyze large datasets
- **SQL –** to write queries and get results from the data
- **Joins** to combine data across multiple tables
- **Aggregate functions** like COUNT, SUM, and GROUP BY
- **Window functions** like ROW_NUMBER ()
- **Filtering and grouping** for clean and meaningful results

📃 Dataset Description:

Table Name	**Description**
brands	List of product brands
categories	Product categories

Table Name	**Description**
customers	Customer information (name, location)
order_items	Line-level details of each order including product, quantity, and price
orders	Orders placed by customers along with order date, staff, and status
products	Product master list including brand and category IDs
staffs	Employees handling customer orders
status_dataset	Possible statuses of each order (e.g., completed, pending, cancelled)
stocks	Inventory levels of products in each store
stores	Retail store details including location

🔄 **Relationships Between Tables:**

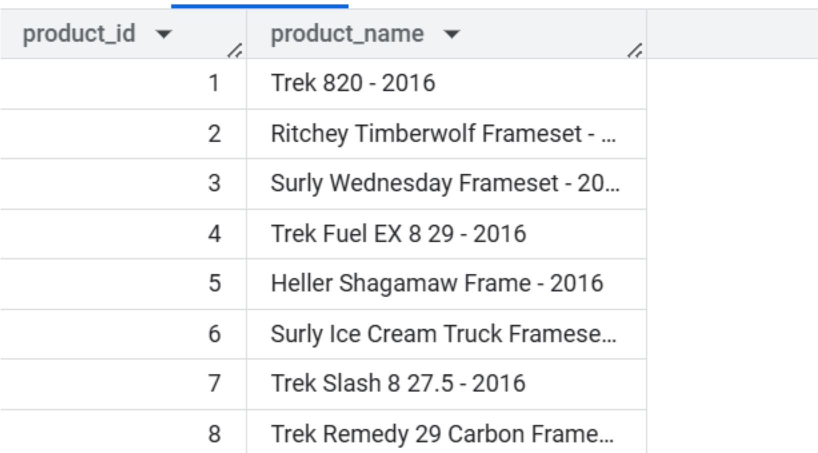
- products ↔ brands & categories (via brand_id, category_id)
- orders ↔ order_items (via order_id)
- orders ↔ staffs (via staff_id)
- orders ↔ customers (via customer_id)
- orders ↔ status_dataset (via status)
- stocks ↔ stores & products (via store_id, product_id)

Problem Statement
Identify product which are not sold to any customer yet. Rejected orders can also be considered as not sold yet.

SQL Query

```
select  distinct p.product_id,p.product_name
from `products.products`p
left join `Order_items.Order_items` oi on p.product_id = oi.product_id
left join `orders.Orders` r
on oi.order_id = r.order_id
and r.order_status = 3 or oi.product_id is null;
```

Sample Output



product_id	product_name
1	Trek 820 - 2016
2	Ritchey Timberwolf Frameset - ...
3	Surly Wednesday Frameset - 20...
4	Trek Fuel EX 8 29 - 2016
5	Heller Shagamaw Frame - 2016
6	Surly Ice Cream Truck Framese...
7	Trek Slash 8 27.5 - 2016
8	Trek Remedy 29 Carbon Frame...

Explanation:

Dataset Used

- products.products: Contains product IDs and names
- Order_items.Order_items: Shows what products were included in orders
- orders.Orders: Contains order status to check if it was completed or rejected

Query Logic

- LEFT JOIN ensures we do not lose any product even if it doesn't exist in orders.
- r.order_status = 3: Includes products in rejected orders
- oi.product_id IS NULL: Includes products that were never ordered
- DISTINCT: Removes duplicates in case the same unsold product appears multiple times

Insight:

This query helps identify **unsold or rejected products**, which can be useful for:

- Product performance review
- Inventory clearance decisions
- Marketing or promotion planning for less popular products

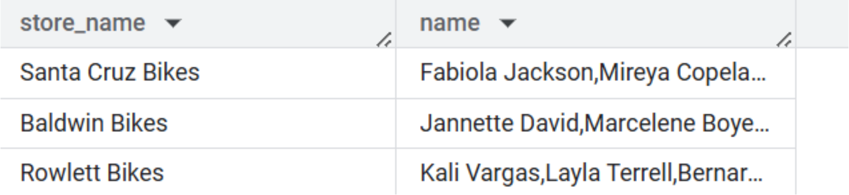
Problem Statement

Display the **store name** and a list of its **employee names** (staff) in a single row per store. Employee names should be **comma-separated** for each store.

SQL Query

```
select s.store_name,string_agg(concat(s2.first_name,' ',s2.last_name),',')as name
from`stores.stores` s
join `Staffs.Staffs` s2 on s.store_id = s2.store_id
group by s.store_name;
```

Sample Output



store_name	name
Santa Cruz Bikes	Fabiola Jackson,Mireya Copela...
Baldwin Bikes	Jannette David,Marcelene Boye...
Rowlett Bikes	Kali Vargas,Layla Terrell,Bernar...

Explanation:

Dataset Used

- stores.stores: Contains store information including store_id and store_name
- Staffs.Staffs: Contains employee details with first_name, last_name, and store_id

Query Logic

- JOIN connects each store to its corresponding employees.
- CONCAT combines first_name and last_name for full names.
- STRING_AGG merges all employee names for the same store into one string.
- The GROUP BY groups the result by store, giving one line per store.

Insights

This output is useful to:

- Quickly view **who works in which store**
- Create **team lists** for store managers

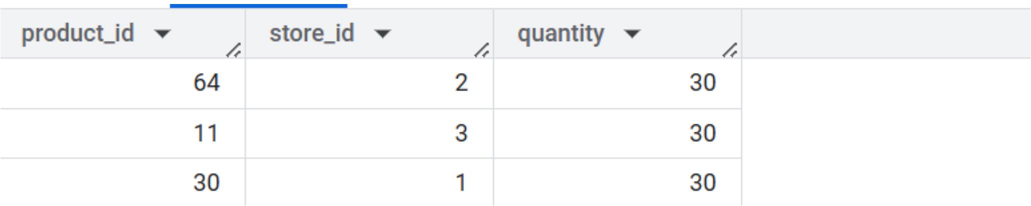
Problem Statement

For each store, find the product that currently has the **highest quantity in stock**.
Display the **product ID**, **store ID**, and the **available quantity**

SQL Query

```
select product_id,store_id,quantity
from(
  select product_id,store_id,quantity,row_number() over(
    partition by store_id order by quantity desc) as rn
    from `stocks.Stocks`
  ) as table
where rn = 1;
```

Sample Output



product_id	store_id	quantity
64	2	30
11	3	30
30	1	30

Explanation:

Dataset Used

- stocks.Stocks: Contains stock levels of each product in each store (fields: store_id, product_id, quantity)

Query Logic

- PARTITION BY store_id: Resets the row number for each store.
- ORDER BY quantity DESC: Ensures that products with the most quantity come first.
- ROW_NUMBER() gives a unique rank to each product within its store group.
- WHERE rn = 1: Filters to keep only the top-ranked product per store.

Insights

- This query helps in:
- Identifying fast-moving vs slow-moving products based on stock levels.
- Understanding inventory strength per store.
- Making restocking decisions by comparing the most stocked items across stores.

Problem Statement

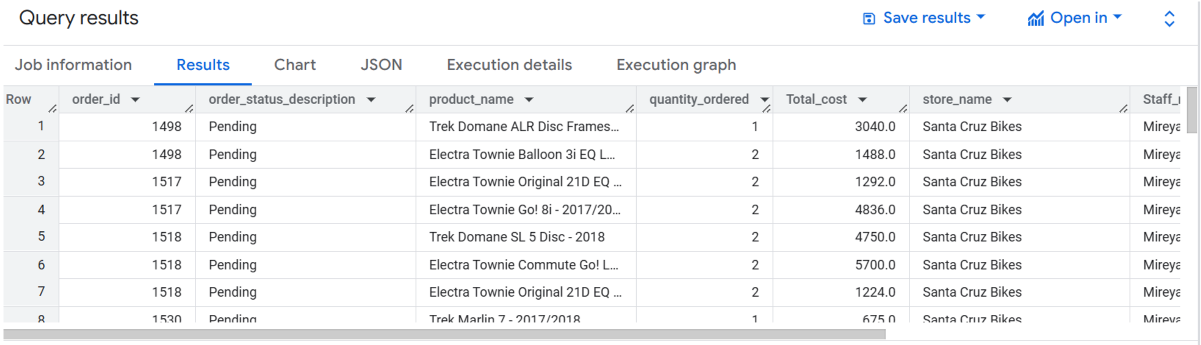
We want to view the **details of all orders** that are **in progress** — meaning they are **neither completed nor rejected**.
The details should include:

- Order ID
- Order status (description)
- Product name
- Quantity ordered
- Total cost
- Store name
- Staff name
- Customer name

SQL Query

```
select r.order_id, s.description as order_status_description, p.product_name, r2.quantity as quantity_ordered, round((r2.quantity*r2.list_price)*(1-r2.discount))as Total_cost, s1.store_name, concat(s2.first_name," ",s2.last_name)as Staff_name, concat(c.first_name," ",c.last_name)as customername

from `orders.Orders` r

join `Order_items.Order_items` r2 on r.order_id = r2.order_id

join `Status_table.Status` s on r.order_status = s.code

join `products.products` p on r2.product_id = p.product_id

join `stores.stores` s1 on r.store_id = s1.store_id

join `Staffs.Staffs` s2 on r.staff_id = s2.staff_id

join `Customerbike.Customer` c on r.customer_id=c.customer_id

where s.description not in("Completed","Rejected");
```

Sample Output



Row	order_id	order_status_description	product_name	quantity_ordered	Total_cost	store_name	Staff_i
1	1498	Pending	Trek Domane ALR Disc Frames...	1	3040.0	Santa Cruz Bikes	Mireya
2	1498	Pending	Electra Townie Balloon 3i EQ L...	2	1488.0	Santa Cruz Bikes	Mireya
3	1517	Pending	Electra Townie Original 21D EQ ...	2	1292.0	Santa Cruz Bikes	Mireya
4	1517	Pending	Electra Townie Go! 8i - 2017/20...	2	4836.0	Santa Cruz Bikes	Mireya
5	1518	Pending	Trek Domane SL 5 Disc - 2018	2	4750.0	Santa Cruz Bikes	Mireya
6	1518	Pending	Electra Townie Commute Go! L...	2	5700.0	Santa Cruz Bikes	Mireya
7	1518	Pending	Electra Townie Original 21D EQ ...	2	1224.0	Santa Cruz Bikes	Mireya
8	1530	Pending	Trek Marlin 7 - 2017/2018	1	675.0	Santa Cruz Bikes	Mireya

Explanation

Datasets Used

- orders.Orders – Order details including order status
- Order_items.Order_items – Product-level details for each order
- Status_table.Status – Description for each order status code
- products.products – Product name
- stores.stores – Store name
- Staffs.Staffs – Staff name
- Customerbike.Customer – Customer name

Query Logic

- **Joins** are used to bring together product, customer, store, and staff details related to each order.
- **Filter**: WHERE s.description NOT IN ("Completed", "Rejected") to keep only active or pending orders.
- **Total Cost** is computed by factoring in quantity, price, and discount.
- CONCAT is used to format full names for staff and customers

Insights

This report is helpful to:

- Track **in-progress or pending orders**.
- Monitor **workload** on staff for unfulfilled orders.
- Analyze the **order pipeline** before completion.
- Identify if there are any delays or issues with current orders

---

Problem Statement

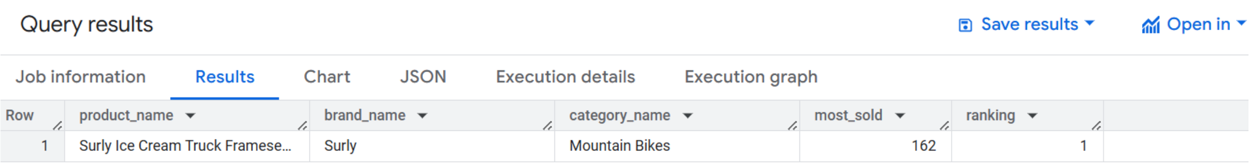
Identify the **product that has been sold the most number of times**, but only from **orders that are marked as completed**.
The result should show:

- Product name
- Brand name
- Category name
- Total quantity sold

SQL Query

```
with cte as (


select p.product_name,b.brand_name,c.category_name, sum(o.quantity) as most_sold
from `Brands.Brands` b
join `products.products`p on b.brand_id = p.brand_id
join `Order_items.Order_items`o on p.product_id = o.product_id
join `Category.Category` c on p.category_id = c.category_id
join `orders.Orders` o1 on o.order_id = o1.order_id
join `Status_table.Status` s on o1.order_status = s.code
where s.description = 'Completed'

group by p.product_name,b.brand_name,c.category_name

order by most_sold desc),


cte1 as (

select c.product_name, c.brand_name, c.category_name, c.most_sold, rank() over(order by most_sold desc) as ranking

from cte c)


select *

from cte1

where ranking = 1;
```

Sample Output



Row	product_name	brand_name	category_name	most_sold	ranking
1	Surly Ice Cream Truck Framese...	Surly	Mountain Bikes	162	1

Explanation

- We **filter only completed orders** to ensure the product was actually sold.
- The query **sums the quantity** of each product across all completed orders.
- Using **RANK()**, we identify the product(s) with the **highest total quantity sold**.
- This method handles **ties** correctly (if two products are tied for top sales).

Datasets Used

- Brands.Brands – Brand information

- products.products – Product details
- Order_items.Order_items – Product quantities
- Category.Category – Category names
- orders.Orders – Order data with status
- Status_table.Status – Status descriptions like "Completed"

Query Logic

1. Use **joins** to combine relevant product, brand, category, and order data.
2. **Filter** only Completed orders using the status description.
3. **Group** by product, brand, and category to calculate total quantity sold.
4. Use **RANK()** to find the top-selling product(s).

Insights

- This helps businesses identify their **best-performing product**.
- It can guide **inventory management**, **restocking decisions**, and **marketing focus**.
- The use of window functions ensures the result is scalable and flexible.

Problem Statement

Find all staff members who have sold more than 1000 products in total.
Each product sold (across any order) should be counted, regardless of order size or status.

SQL Query

```
select s.staff_id,concat(s.first_name," ",s.last_name) as full_name,

count(i.product_id) as product_sold_count

from `Staffs.Staffs` s

join `orders.Orders` o on s.staff_id=o.staff_id

join `Order_items.Order_items` i on o.order_id = i.order_id
```

```
group by s.staff_id,s.first_name,s.last_name

having count(i.product_id)>1000;
```

Explanation

- We **join the staff table with orders and order items** to trace who sold which products.
- COUNT(i.product_id) calculates the total number of **individual product sales** linked to each staff member.
- Using HAVING, we **filter out staff who have sold 1000 or fewer products**.

Datasets Used

- Staffs.Staffs – Contains staff information
- orders.Orders – Links staff to orders
- Order_items.Order_items – Details of products sold in each order

Query Logic

1. Join **Staffs → Orders → Order Items** to associate staff with products sold.
2. Use COUNT(product_id) to **sum up how many products** each staff member sold.
3. Filter results to **only show those with over 1000 sales**.

Insights

- Highlights **top-performing sales staff**.
- Useful for **performance reviews**, **incentive planning**, or **recognition**.
- Businesses can use this to reward or promote staff with high contributions.

🔷 Conclusion

This project demonstrates how **BigQuery SQL** can be used to extract meaningful business insights from complex relational data. By analyzing 10 interconnected datasets from a retail environment, we were able to:

- Identify unsold products, helping reduce overstock.

- Map staff distribution across stores for better team visibility.
- Highlight inventory patterns to optimize stock levels per store.
- Monitor active orders and track progress for operational efficiency.
- Determine top-selling products to improve marketing and demand forecasting.
- Recognize high-performing staff to support reward programs or performance reviews.

With SQL queries, we turned raw data into **actionable insights** that can support better decision-making in sales, staffing, inventory, and customer service strategies.

This project reinforces the value of **data-driven thinking** in a retail context and showcases how powerful insights can be extracted with the right SQL logic and data relationships.