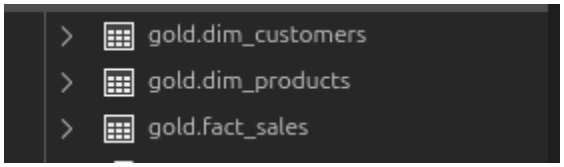
Hi there,
I'm doing a project to find the insights from the data related to customers, products, sales etc.
Let's do it together!!

i use 3 files here, which I took from GitHub:
These are the tables



Microsoft SQL Server Management Studio (SSMS) is a Windows-only application and does not have native support for Linux-based operating systems like Ubuntu. Since I am using Ubuntu 24 LTS, I had to find an alternative to SSMS to interact with my SQL Server instance.

Alternative to SSMS on Ubuntu
To manage SQL Server on Ubuntu, I used Azure Data Studio, which is a cross-platform tool that supports Windows, macOS, and Linux. Unlike SSMS, Azure Data Studio provides a modern UI, integrated terminal, and support for Jupyter notebooks, making it suitable for SQL development and analytics.

If I needed more command-line interaction, I also used sqlcmd, which allowed me to run SQL queries from the terminal.

Setting Up SQL Server on Docker and Azure Data Studio on Ubuntu

1 Install Docker on Ubuntu
First, update your package lists and install Docker:

```
sudo apt update
sudo apt install -y docker.io
```

Enable and start the Docker service:

```
sudo systemctl enable --now docker
```

Verify that Docker is running:

```
docker --version
```

2. Download & Run SQL Server in Docker

Pull the latest SQL Server image from Microsoft:

```
sudo docker pull mcr.microsoft.com/mssql/server:latest
```

Now, create and start a SQL Server container:

```
sudo docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=YourStrongPassword123' \
   -p 1433:1433 --name sql_server_container \
   -d mcr.microsoft.com/mssql/server:latest
```

Verify if the container is running:

```
sudo docker ps -a
```

3. Install Azure Data Studio

Download Azure Data Studio:

```
wget https://aka.ms/azuredatastudio-linux -O azuredatastudio.deb
```

Then install it using dpkg:

```
sudo apt install -y ./azuredatastudio.deb
```

Run Azure Data Studio:

```
azuredatastudio
```

4. Connect to SQL Server in Azure Data Studio

- Open Azure Data Studio
- Click New Connection
- Enter the following details:
- Server Name → `localhost,1433`
- Authentication Type → `SQL Login`
- Username → `sa`
- Password → `YourStrongPassword123`
- **Click Connect**

5. Connect Using sqlcmd in Terminal

Run:

```
sudo docker exec -it sql_server_container /opt/mssql-tools/bin/sqlcmd -S localhost
-U sa -P "YourStrongPassword123"
```

Test connection:

```
SELECT name FROM sys.databases;
GO
```

6. Copy CSV Files into the Container

If you need to import data from CSV files into SQL Server:

```
sudo docker cp /home/himanshu/Desktop/Data\ Scientist\
Resources/Data/sql-data-analytics-project-main/datasets/csv-files/gold.dim_customer
s.csv sql_server_container:/var/opt/mssql/data/
```

7. Grant Permissions to SQL Server for Bulk Load

```
sudo docker exec -it sql_server_container chmod 777
/var/opt/mssql/data/gold.dim_customers.csv
```

8. Load CSV into SQL Server

Inside sqlcmd, run:

```
BULK INSERT gold.dim_customers
FROM '/var/opt/mssql/data/gold.dim_customers.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n'
);
GO
```

CODE:

1- Trends(Change over Time)

```
SELECT
DATETRUNC(month, order_date) AS order_date,
SUM(sales_amount) AS order_month,
COUNT(DISTINCT customer_key) AS total_customers,
SUM(quantity) AS total_quantity

FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(MONTH,order_date)
ORDER BY DATETRUNC(MONTH, order_date)
```

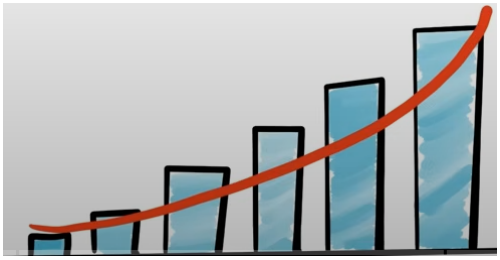
DATETRUNC(): Round a date or timestamp to a specified date part.

```
DATETRUNC(month, order_date) AS order_date,
```

This will return the **first day of the month for each order_date**.

2- Cumulative Analysis(Aggregate the data progressively over the time)

It helps us to understand whether our business is growing or declining.



It helps us to check our business is growing or declining.

```
SELECT
order_date,
total_sales,
--window funtion
SUM(total_sales) OVER(ORDER BY order_date) as running_total_sales
FROM
(
SELECT
```

```
DATETRUNC(MONTH, order_date) AS order_date,
SUM(sales_amount) AS total_sales
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(MONTH, order_date)
--ORDER BY DATETRUNC(MONTH, order_date)
) t
```

Adding each value with the previous sales_value.

Results Messages

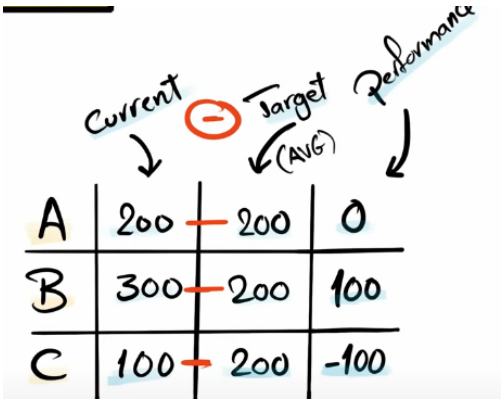
	order_date	total_sales	running_total_sales
1	2010-12-01	43419	43419
2	2011-01-01	469795	513214
3	2011-02-01	466307	979521
4	2011-03-01	485165	1464686
5	2011-04-01	502042	1966728
6	2011-05-01	561647	2528375
7	2011-06-01	737793	3266168
8	2011-07-01	596710	3862878
9	2011-08-01	614516	4477394
10	2011-09-01	603047	5080441
11	2011-10-01	708164	5788605
12	2011-11-01	660507	6449112
13	2011-12-01	669395	7118507
14	2012-01-01	495363	7613870

Here is default window frame(between Unbound preceding and current row)

```
SELECT
order_date,
total_sales,
--window funtion
SUM(total_sales) OVER(PARTITION BY order_date ORDER BY order_date) AS
running_total_sales,
AVG(avg_price) OVER (PARTITION BY order_date ORDER BY order_date) AS moving_avg_price
FROM
(
SELECT
DATETRUNC(MONTH, order_date) AS order_date,
SUM(sales_amount) AS total_sales,
AVG(price) AS avg_price
FROM gold.fact_sales
WHERE order_date IS NOT NULL
GROUP BY DATETRUNC(MONTH, order_date)
--ORDER BY DATETRUNC(MONTH, order_date)
) t
```

Now we will do Performance analysis:

Basically performance value means comparing current value with target value. It helps us to compare success and compare performance.



Current[measure] - Target[measure]

Current sales - target sales
Current year sales - previous year sales
Current sales - lowest sales

```sql
SELECT
YEAR(f.order_date) AS order_year,
p.product_name,
SUM(f.sales_amount) AS current_sales
FROM gold.fact_sales as f
LEFT JOIN gold.dim_products p
ON f.product_key = p.product_key
WHERE order_date IS NOT NULL
GROUP BY YEAR(f.order_date),
p.product_name
```

Based on the output of this querry we will do aggrigations and calculation.

Results Messages

	order_year	product_name	current_sales
1	2012	Mountain-200 Black- 38	342921
2	2012	Touring Tire Tube	10
3	2011	Mountain-100 Silver- 44	156400
4	2013	Long-Sleeve Logo Jersey- L	21550
5	2013	Racing Socks- L	2340
6	2014	LL Road Tire	1176
7	2014	Mountain Tire Tube	830
8	2011	Road-550-W Yellow- 38	1000
9	2011	Road-150 Red- 44	1001840
10	2013	Half-Finger Gloves- L	10248
11	2013	Mountain-200 Black- 46	947835
12	2014	Women's Mountain Shorts- L	1610
13	2013	Long-Sleeve Logo Jersey- S	20350
14	2011	Road-250 Red- 44	4886

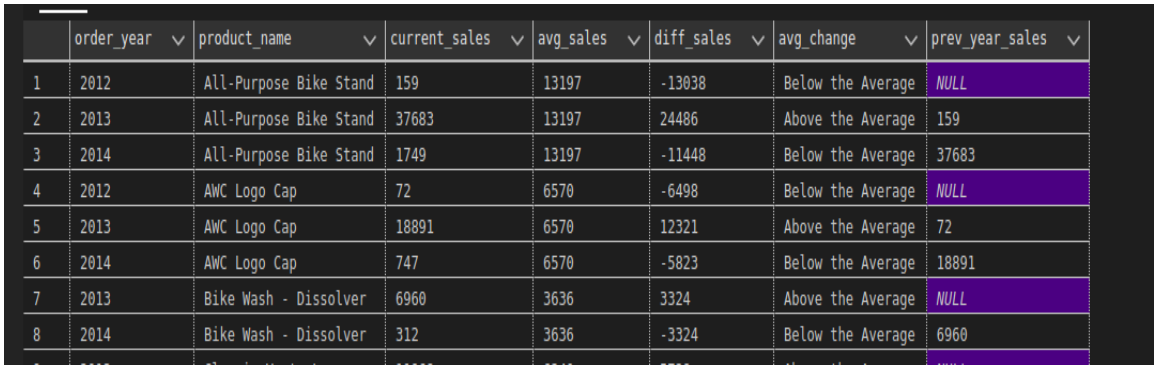
Analyzing the yearly performance of the products by comparing each product's sales to both its average sales performance and the previous year sales.

To find which product is below then avg or above then avg.
To get previous year sales so we can calculate sales performance accordingly.

```
WITH yearly_product_sales AS (

SELECT
YEAR(f.order_date) AS order_year,
p.product_name,
SUM(f.sales_amount) AS current_sales
FROM gold.fact_sales as f
LEFT JOIN gold.dim_products p
ON f.product_key = p.product_key
WHERE order_date IS NOT NULL
GROUP BY YEAR(f.order_date),
p.product_name
)

SELECT
order_year,product_name,current_sales,
AVG(current_sales) OVER(PARTITION BY product_name) AS avg_sales,
current_sales - AVG(current_sales) OVER(PARTITION BY product_name) AS diff_sales,
-- i want to make kinda flag that is it above the average or below the avg
CASE WHEN (current_sales - AVG(current_sales) OVER(PARTITION BY product_name)) > 0
THEN 'Above the Average'
   WHEN  (current_sales - AVG(current_sales) OVER(PARTITION BY product_name)) < 0 THEN
'Below the Average'
   ELSE 'avg'
END avg_change,
LAG(current_sales) OVER(PARTITION BY  product_name ORDER BY order_year) AS
prev_year_sales
FROM yearly_product_sales
ORDER BY product_name, order_year
```



	order_year	product_name	current_sales	avg_sales	diff_sales	avg_change	prev_year_sales
1	2012	All-Purpose Bike Stand	159	13197	-13038	Below the Average	NULL
2	2013	All-Purpose Bike Stand	37683	13197	24486	Above the Average	159
3	2014	All-Purpose Bike Stand	1749	13197	-11448	Below the Average	37683
4	2012	AWC Logo Cap	72	6570	-6498	Below the Average	NULL
5	2013	AWC Logo Cap	18891	6570	12321	Above the Average	72
6	2014	AWC Logo Cap	747	6570	-5823	Below the Average	18891
7	2013	Bike Wash - Dissolver	6960	3636	3324	Above the Average	NULL
8	2014	Bike Wash - Dissolver	312	3636	-3324	Below the Average	6960

Year-over-Year analysis

```
---year over year analysis
   LAG(current_sales) OVER(PARTITION BY  product_name ORDER BY order_year) AS
prev_year_sales,
   current_sales - LAG(current_sales) OVER(PARTITION BY  product_name ORDER BY
order_year) AS Yearly_diff_sales,
   CASE WHEN current_sales - LAG(current_sales) OVER(PARTITION BY  product_name ORDER
BY order_year) >0 THEN 'Increasing'
        WHEN current_sales - LAG(current_sales) OVER(PARTITION BY  product_name ORDER
BY order_year) < 0 THEN 'Decreasing'
        ELSE 'No change'
   END yearly_status
```

Year-over-Year (YoY) analysis is used to compare business performance, trends, and growth over time by comparing the same period in different years.

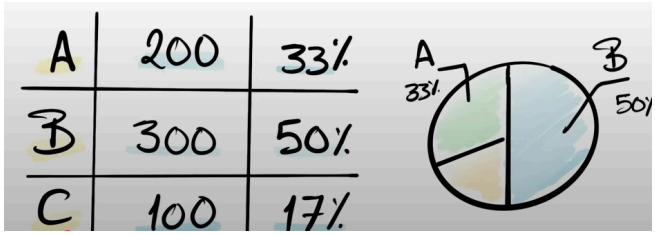
prev_year_sales	Yearly_diff_sales	yearly_status
NULL	NULL	No change
159	37524	Increasing
37683	-35934	Decreasing
NULL	NULL	No change
72	18819	Increasing
18891	-18144	Decreasing
NULL	NULL	No change
6960	-6648	Decreasing
NULL	NULL	No change
11968	-11456	Decreasing
NULL	NULL	No change
11840	-10944	Decreasing
NULL	NULL	No change

There are several reasons for YoY analysis:

- Remove Seasonality Effects
- Track business growth and trends
- Identifies long term performance
- Useful for forecasting and decision-making
- Detects Anomalies and Market Changes

Part to Whole Analysis:

Analyze how an individual part is performing compared to overall, allowing us to understand which category has the greatest impact on our business.



[Measure] / Total[Measure] * 100 by [Dimension]

(sales/ total sales) * 100 by category
(quantity / total quantity) * 100 by country

Note: To display aggregations at the multiple levels in the results, use window function.

```sql
WITH category_sales AS (
SELECT
category,
SUM(sales_amount) tot_sales
from gold.fact_sales f
LEFT JOIN gold.dim_products p
ON f.product_key = p.product_key
GROUP BY category
)
--To display aggrigations at the multiple levels in the results, use window function.
SELECT
category,
tot_sales,
SUM(tot_sales) OVER() overall_sales,
CONCAT(ROUND((CAST (tot_sales AS FLOAT) / SUM(tot_sales) OVER()) * 100, 2), '%') AS
percent_overall_Sales
FROM category_sales
ORDER BY tot_sales DESC
```

Here we can say that,
96% of total sales done with Bikes in business.

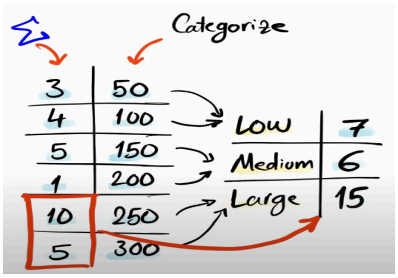
Results | Messages

	category	tot_sales	overall_sales	percent_overall_Sales
1	Bikes	28316272	29356250	96.46%
2	Accessories	700262	29356250	2.39%
3	Clothing	339716	29356250	1.16%

We can use orders or total numbers of customers if we want more clarification about our business or sales.

Next step is **Data Segmentation**

Here we will group the data based on specific ranges. To help us understand the correlation between two measures.

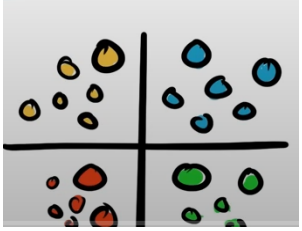


[Measure] by [Measure]
Total product by sales range
Total customers by Age

Data Segmentation is the process of dividing a dataset into smaller, meaningful groups based on certain criteria. This helps in better analysis, decision-making, and targeting specific subsets of data.

It is important for:

- Better insights and analysis
- Improved decision making



- More effective marketing and personalization
    - Used in targeted advertising, recommending **relevant products** to specific customer segments.
- Optimize resource allocation

Here we'll **categorize products** into different cost ranges and count how many products fall into each category.

```
WITH prod_segments AS (
SELECT
product_key,
category,
cost,
CASE WHEN cost <500 THEN '1-500'
   WHEN cost>500 AND cost<800 THEN '500-800'
   WHEN cost>800 AND cost<1000 THEN '800-1000'
   ELSE '1000+'
END cost_range
FROM gold.dim_products
)
SELECT
cost_range,
COUNT(product_key) AS total_products
FROM prod_segments
GROUP BY cost_range
ORDER BY cost_range ASC
```

--Now we will group customers into 3 segments beased on their spending behavior.

-- VIP: Atlest 12 months of history and spending more then 5000$
-- Regular: Atlest 12 months of history and spending more then 5000$ or less
-- New: lifespan less then 12 Months

-- and we'll find the total customers in each category

```
DATEDIFF(MONTH, MIN(order_date), MAX(order_date))
```

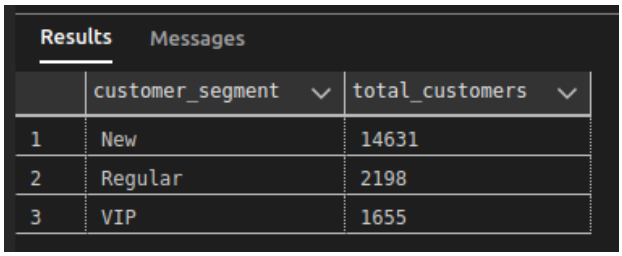
DATEDIFF(): This function calculates the difference between two dates in the specific unit (years, months, days etc)

DATEDIFF(interval, start_date, end_date)

```
WITH Customer_spending AS(
```

```sql
SELECT
c.customer_key,
SUM(f.sales_amount) AS total_spending,
--we need to check first order and last order to check how many months are in between.
MIN(order_date) AS first_date,
MAX(order_date) AS last_date,
DATEDIFF(MONTH, MIN(order_date), MAX(order_date)) AS lifespan
FROM gold.fact_sales f
LEFT JOIN gold.dim_customers c
ON f.customer_key = c.customer_key
GROUP BY c.customer_key
)
SELECT
customer_segment,
COUNT(customer_key) AS total_customers
FROM (
   SELECT
   customer_key,
   CASE
        WHEN total_spending > 5000 AND lifespan >= 12 THEN 'VIP'
        WHEN total_spending <=5000 AND lifespan>=12 THEN 'Regular'
        ELSE 'New'
   END AS customer_segment
   FROM Customer_spending ) t
GROUP BY customer_segment
ORDER BY total_customers DESC
```

Got amazing segmentation.



Results | Messages

	customer_segment	total_customers
1	New	14631
2	Regular	2198
3	VIP	1655

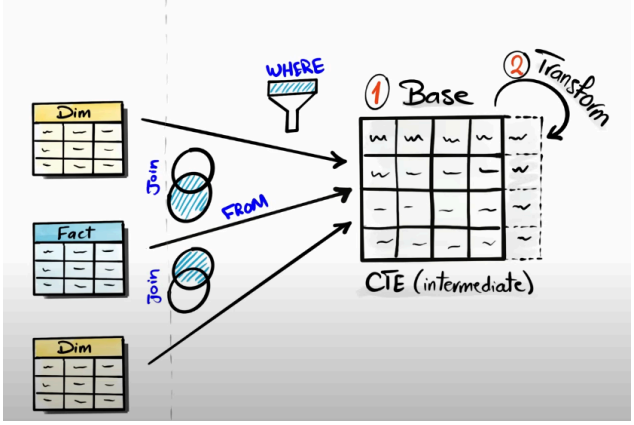
Creating Report

Purpose- This report consolidates key customer metrics and behavior.

Highlights:

1. Gathers essential fields such as names, ages, and transaction details.
2. Segments customers into categories (VIP, Regular, New) and age groups.
3. Aggregates customer-level metrics:
    - total orders
    - total sales
    - total quantity purchased
    - total products
    - lifespan (in months)
4. Calculates valuable KPIs:
    - recency (months since last order)
    - average order value
    - average monthly spend

Here I will create a Base table by using joins to all 3 tables. Instead of copying all the columns, using Joins I'll select specific columns from the tables and create an Intermediate table(CTE) for our query output.



Created Report table,

Now, we will create a **view table** for our **report table** so that other developers can easily access the required data without dealing with complex queries.

A **view** in SQL Server is a **virtual table** that is created based on the result of a SQL query. Unlike a physical table, a view **does not store data** permanently instead, it dynamically fetches data from the underlying tables whenever it is queried.