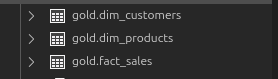
Hi there,

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**

**1️ 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 |
| --- |

**2️. 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 |
| --- |

**3️. 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 |
| --- |

**4️. 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**

**5️. 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 |
| --- |

**6️. 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\_customers.csv sql\_server\_container:/var/opt/mssql/data/ |
| --- |

**7️. 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 |
| --- |

**8️. 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)

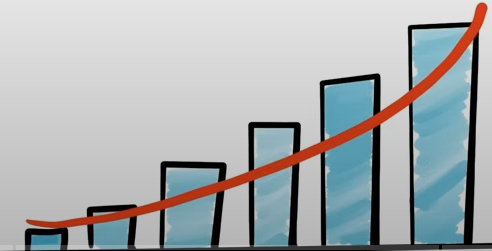
**DATETRUNC():** Round a date or timestamp to a specified date part.

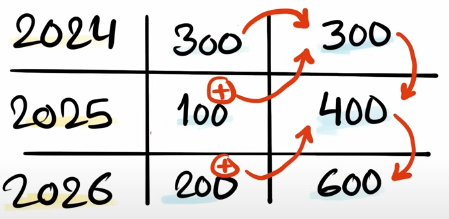
DATETRUNC(month, order\_date) AS order\_date,

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

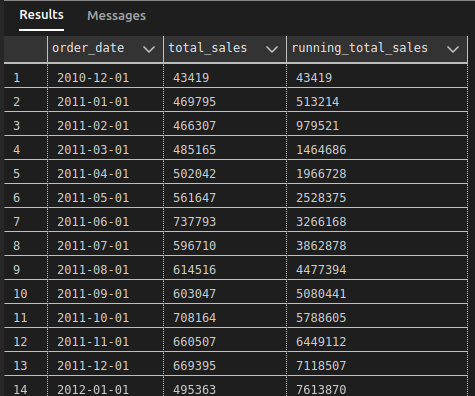
WHERE order\_date IS NOT NULL

GROUP BY DATETRUNC(MONTH, order\_date)

--ORDER BY DATETRUNC(MONTH, order\_date)

) t

Adding each value with the previous sales\_value.



**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**

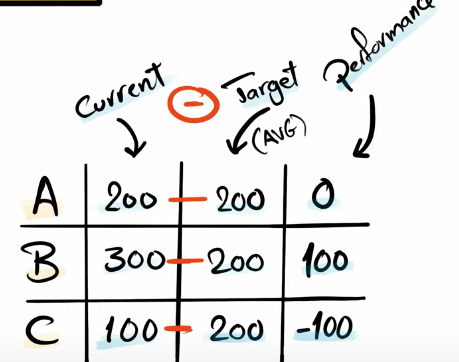
**WHERE order\_date IS NOT NULL**

**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

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

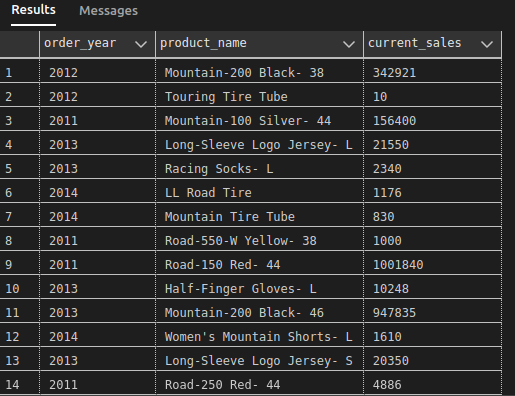
ON f.product\_key = p.product\_key

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.



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'

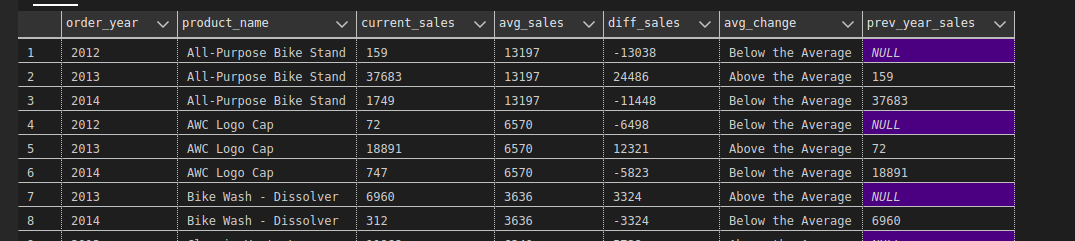
ELSE 'avg'

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

****

**Year-over-Year analysis**

**---year over year analysis**

**LAG(current\_sales) OVER(PARTITION BY product\_name ORDER BY order\_year) AS prev\_year\_sales,**

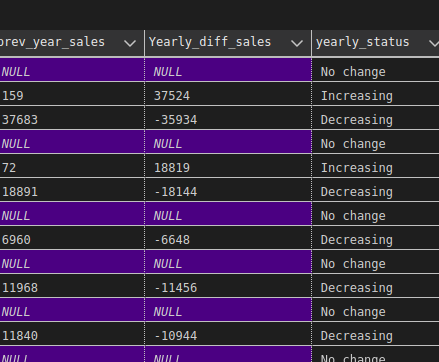
**current\_sales - LAG(current\_sales) OVER(PARTITION BY product\_name ORDER BY order\_year) AS Yearly\_diff\_sales,**

**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**

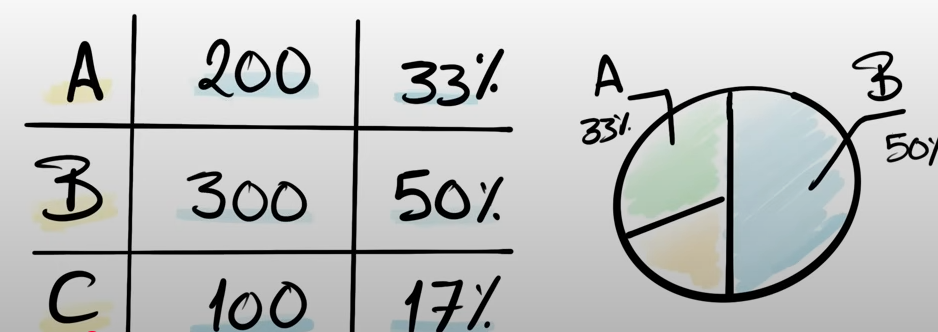


**Year-over-Year (YoY) analysis is used to compare business performance, trends, and growth over time by comparing the same period in different years.**

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.

**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,**

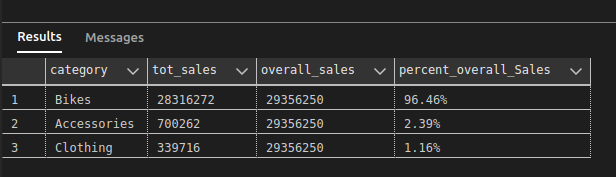
**tot\_sales,**

**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**

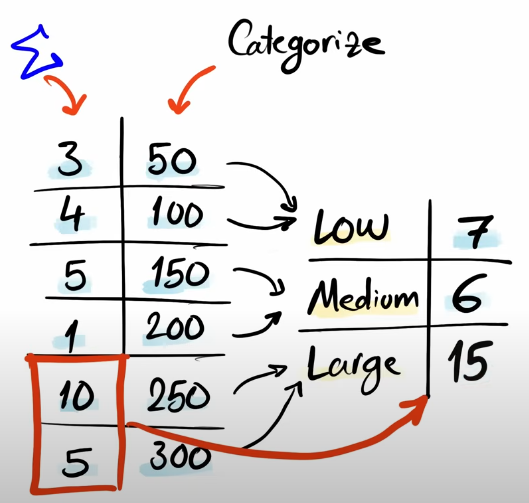


Here we can say that,

96% of total sales done with Bikes in business.

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

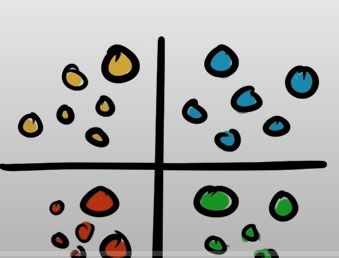
Next step is **Data Segmentation**

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(

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'

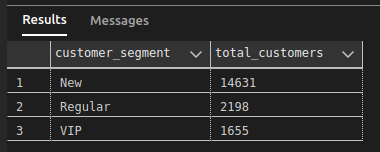
END AS customer\_segment

FROM Customer\_spending ) t

GROUP BY customer\_segment

ORDER BY total\_customers DESC

Got amazing segmentation.



**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

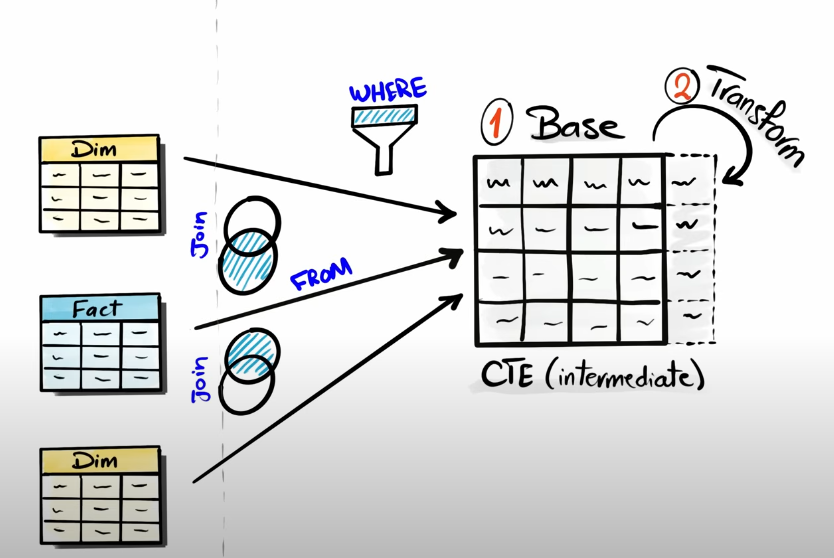
- total products

- 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.