

**Documentation on**  
**SuperStore Sales and Profit**  
**Analysis Report**  
**By**  
**Yash Srivastava**

# **Index**

1. About the project
2. Problem faced in the project
3. Cleaning the dataset
4. Analyze the dataset
5. Visualization of dataset
6. Interpretation of data

## **About the Project**

The data on which I did a project is of the “SuperStore Sales and Profit ” from 2019 to 2020. This report shows SuperStore's sales across all the states of the United States. This report records the sales for different items for one year from 2019 to 2020 (Furniture, Technology, office supplies). We use various software, ways, and functions to analyze the data and make it easy to read and understand by cleaning and summarizing it.

I'm using the tools Ms.Excel, MySQL, Python (Anaconda Navigator) in Jupyter Notebook, Power BI, and Tableau to clean, analyze, interpret, and summarize the data.

## **Problem Faced in the Project**

Before cleaning the data in Excel, we tried uploading the dataset in SQL but found that only 2% and 1% of the data was uploaded.

After many attempts, we found that my data needed to be clearer. We cleaned the data in MS Excel and found that the data format needed to be more organized. We changed all data types in columns like strings into text and int into numeric.

Again, we uploaded the data in SQL, and we found that only 170 rows were uploaded out of 5707 rows. After a few more adjustments, we again uploaded the data in SQL, so we found only 175 rows uploaded in SQL.

Finally, after many attempts, we were able to upload our entire data set, which has 5707 rows and 21 columns.

## Working on the project

### 1. Cleaning of dataset(MS Excel)

FileHomeInsertPage LayoutFormulasDataReviewViewHelp																	CommentsShare		
Clipboard			Font			Alignment			Number			Styles		Cells		Editing			Add-ins
<div>PasteCutCopyFormat Painter</div>			<div>Calibri11A<sup>+</sup>A<sup>-</sup>BBIU</div>			<div>Wrap TextMerge &amp; Center</div>			<div>Text%</div>			<div>Conditional FormattingFormat as TableCell Styles</div>		<div>InsertDeleteFormat</div>		<div>AutoSumFillSort &amp; FilterFind &amp; Select</div>			<div>Add-ins</div>
E11Standard Class																			
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O				
1	Row id	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	State	Region	Product ID	Category	Sub-Category	Product Name			
2	4918	CA-2019-160304	2019-01-01	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	Maryland	East	FUR-BO-10004709	Furniture	Bookcases	Bush Westfield C			
3	4919	CA-2019-160304	2019-01-02	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	Maryland	East	FUR-BO-10004709	Furniture	Bookcases	Bush Westfield C			
4	4920	CA-2019-160304	2019-01-02	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	Maryland	East	TEC-PH-10000455	Technology	Phones	GE 30522EE2			
5	3074	CA-2019-125206	2019-01-03	2019-01-05	First Class	LR-16915	Lena Radford	Consumer	United States	Los Angeles	California	West	OFF-ST-10003692	Office Supplies	Storage	Recycled Steel Pe			
6	8604	US-2019-116365	2019-01-03	2019-01-08	Standard Class	CA-12310	Christine Abelman	Corporate	United States	San Antonio	Texas	Central	TEC-AC-10002217	Technology	Accessories	Imation Clip USB			
7	8605	US-2019-116365	2019-01-03	2019-01-08	Standard Class	CA-12310	Christine Abelman	Corporate	United States	San Antonio	Texas	Central	TEC-AC-10002942	Technology	Accessories	WD My Passport			
8	8606	US-2019-116365	2019-01-03	2019-01-08	Standard Class	CA-12310	Christine Abelman	Corporate	United States	San Antonio	Texas	Central	TEC-PH-10002890	Technology	Phones	AT&T 17929 Lens			
9	9494	CA-2019-105207	2019-01-03	2019-01-08	Standard Class	BO-11350	Bill Overfelt	Corporate	United States	Broken Arrow	Oklahoma	Central	FUR-TA-10000617	Furniture	Tables	Hon Practical Foi			
10	9495	CA-2019-105207	2019-01-03	2019-01-08	Standard Class	BO-11350	Bill Overfelt	Corporate	United States	Broken Arrow	Oklahoma	Central	OFF-BI-10004364	Office Supplies	Binders	Storex Dura Pro I			
11	2898	US-2019-164630	2019-01-04	2019-01-09	Standard Class	EB-13975	Erica Bern	Corporate	United States	Charlotte	North Carolina	South	TEC-CO-10000971	Technology	Copiers	Hewlett Packard			
12	5868	CA-2019-158211	2019-01-04	2019-01-08	Standard Class	BP-11185	Ben Peterman	Corporate	United States	Philadelphia	Pennsylvania	East	OFF-AR-10004078	Office Supplies	Art	Newell 312			
13	5869	CA-2019-158211	2019-01-04	2019-01-08	Standard Class	BP-11185	Ben Peterman	Corporate	United States	Philadelphia	Pennsylvania	East	OFF-BI-10002026	Office Supplies	Binders	Avery Arch Ring E			
14	863	CA-2019-134474	2019-01-05	2019-01-07	Second Class	AJ-10795	Anthony Johnson	Corporate	United States	Jacksonville	Florida	South	TEC-AC-10001714	Technology	Accessories	Logitech MX Perf			
15	864	CA-2019-134474	2019-01-05	2019-01-07	Second Class	AJ-10795	Anthony Johnson	Corporate	United States	Jacksonville	Florida	South	OFF-AR-10003958	Office Supplies	Art	Newell 337			
16	865	CA-2019-134474	2019-01-05	2019-01-07	Second Class	AJ-10795	Anthony Johnson	Corporate	United States	Jacksonville	Florida	South	TEC-PH-10002923	Technology	Phones	Logitech B530 US			
17	2162	CA-2019-101938	2019-01-07	2019-01-12	Standard Class	DW-13480	Dianna Wilson	Home Office	United States	Oakland	California	West	OFF-AR-10003696	Office Supplies	Art	Panasonic KP-35i			
18	8031	CA-2019-158806	2019-01-07	2019-01-11	Standard Class	NM-18520	Neoma Murray	Consumer	United States	Amarillo	Texas	Central	FUR-FU-10004270	Furniture	Furnishings	Executive Impres			
19	8032	CA-2019-158806	2019-01-07	2019-01-11	Standard Class	NM-18520	Neoma Murray	Consumer	United States	Amarillo	Texas	Central	OFF-PA-10004621	Office Supplies	Paper	Xerox 212			
20	6851	US-2019-100461	2019-01-08	2019-01-12	Standard Class	JO-15145	Jack O'Briant	Corporate	United States	Franklin	Wisconsin	Central	FUR-BO-10002545	Furniture	Bookcases	Atlantic Metals M			
21	6852	US-2019-100461	2019-01-08	2019-01-12	Standard Class	JO-15145	Jack O'Briant	Corporate	United States	Franklin	Wisconsin	Central	OFF-BI-10001460	Office Supplies	Binders	Plastic Binding Cc			
22	7808	US-2019-137295	2019-01-08	2019-01-13	Standard Class	VS-21820	Vivek Sundaresam	Consumer	United States	Raleigh	North Carolina	South	OFF-BI-10004236	Office Supplies	Binders	XtraLife ClearVue			
23	7809	US-2019-137295	2019-01-08	2019-01-13	Standard Class	VS-21820	Vivek Sundaresam	Consumer	United States	Raleigh	North Carolina	South	OFF-AR-10001955	Office Supplies	Art	Newell 319			
24	7810	US-2019-137295	2019-01-08	2019-01-13	Standard Class	VS-21820	Vivek Sundaresam	Consumer	United States	Raleigh	North Carolina	South	TEC-PH-10004080	Technology	Phones	Avaya 5410 Digit			
25	3209	CA-2019-108882	2019-01-09	2019-01-15	Standard Class	LA-16780	Laura Armstrong	Corporate	United States	Fresno	California	West	TEC-AC-10000420	Technology	Accessories	Logitech G500s L			
26	3210	CA-2019-108882	2019-01-09	2019-01-15	Standard Class	LA-16780	Laura Armstrong	Corporate	United States	Fresno	California	West	TEC-PH-10002726	Technology	Phones	netTALK DUO Vo			
27	3702	CA-2019-126543	2019-01-09	2019-01-13	Second Class	MF-17665	Maureen Fritzier	Corporate	United States	Toledo	Ohio	East	FUR-FU-10002445	Furniture	Furnishings	DAX Two-Tone R			

Firstly we have to open our dataset in MS Excel to clean and fix the data types in the data with the suitable datatypes.

The total rows= 5707

The total columns= 21

Like:

String into text

Int into numeric

Date into a long date

We have converted the date format which is suitable as per the SQL date format.

Before changes

C	D
Order Date	Ship Date
01-01-2019	07-01-2019
02-01-2019	07-01-2019
02-01-2019	07-01-2019
03-01-2019	05-01-2019
03-01-2019	08-01-2019
03-01-2019	08-01-2019
03-01-2019	08-01-2019
03-01-2019	08-01-2019
03-01-2019	08-01-2019
04-01-2019	09-01-2019
04-01-2019	08-01-2019
04-01-2019	08-01-2019
05-01-2019	07-01-2019
05-01-2019	07-01-2019
05-01-2019	07-01-2019
07-01-2019	12-01-2019
07-01-2019	11-01-2019
07-01-2019	11-01-2019
08-01-2019	12-01-2019
08-01-2019	12-01-2019
08-01-2019	13-01-2019
08-01-2019	13-01-2019
08-01-2019	13-01-2019
09-01-2019	15-01-2019
09-01-2019	15-01-2019
09-01-2019	13-01-2019

After changes

Order Date	Ship Date
2019-01-01	2019-01-07
2019-01-02	2019-01-07
2019-01-02	2019-01-07
2019-01-03	2019-01-05
2019-01-03	2019-01-08
2019-01-03	2019-01-08
2019-01-03	2019-01-08
2019-01-03	2019-01-08
2019-01-04	2019-01-09
2019-01-04	2019-01-08
2019-01-04	2019-01-08
2019-01-05	2019-01-07
2019-01-05	2019-01-07
2019-01-05	2019-01-07
2019-01-07	2019-01-12
2019-01-07	2019-01-11
2019-01-07	2019-01-11
2019-01-08	2019-01-12
2019-01-08	2019-01-12
2019-01-08	2019-01-13
2019-01-08	2019-01-13
2019-01-08	2019-01-13
2019-01-09	2019-01-15
2019-01-09	2019-01-15
2019-01-09	2019-01-13

After studying and knowing about the data we move further to analyze the data.

## 2. Analyze the dataset(My SQL)

We are moving further into the second step of analyzing the dataset we have taken with the help of MySQL.

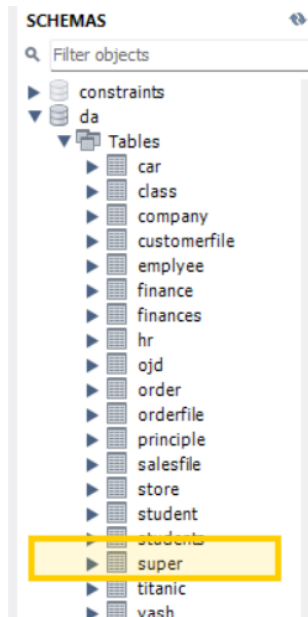
Now, we have loaded the clean data which we have cleaned using MS Excel.

The screenshot displays a MySQL query editor interface. The top toolbar includes icons for file operations, execution, and a 'Limit to 1000 rows' dropdown. The query editor contains the following SQL code:

```
1 • use da
2 select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9 select month(`order date`) from super
10
```

Below the query editor is the 'Result Grid' section, which includes a 'Filter Rows' input and an 'Exports' button. The grid displays a table with 16 columns: Row id, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, City, State, Region, Product ID, Category, Sub-Category, and Price. The first 10 rows of data are visible:

Row id	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	State	Region	Product ID	Category	Sub-Category	Price
1	CA-2019-152156	2019-11-08	2019-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	Kentucky	South	FUR-BO-10001798	Furniture	Bookcases	Bus
2	CA-2019-152156	2019-11-08	2019-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	Kentucky	South	FUR-CH-10000454	Furniture	Chairs	Hor
3	CA-2019-138688	2019-06-12	2019-06-16	Second Class	DV-13045	Darrin Van Huff	Corporate	United States	Los Angeles	California	West	OFF-LA-10000240	Office Supplies	Labels	Self
13	CA-2020-114412	2020-04-15	2020-04-20	Standard Class	AA-10480	Andrew Allen	Consumer	United States	Concord	North Carolina	South	OFF-PA-10002365	Office Supplies	Paper	Xen
14	CA-2019-161389	2019-12-05	2019-12-10	Standard Class	IM-15070	Irene Maddox	Consumer	United States	Seattle	Washington	West	OFF-BI-10003656	Office Supplies	Binders	Fell
22	CA-2019-137330	2019-12-09	2019-12-13	Standard Class	KB-16585	Ken Black	Corporate	United States	Fremont	Nebraska	Central	OFF-AR-10000246	Office Supplies	Art	Nev
23	CA-2019-137330	2019-12-09	2019-12-13	Standard Class	KB-16585	Ken Black	Corporate	United States	Fremont	Nebraska	Central	OFF-AP-10001492	Office Supplies	Appliances	Acc
74	US-2020-156090	2020-07-16	2020-07-18	Second Class	SE-20065	Sandra Flanagan	Consumer	United States	Philadelphia	Pennsylvania	East	FI-ID-10007774	Furniture	Chairs	Clw



We have loaded the data by writing the query:

### **(i). Query: Fetch all rows (limit 6000)**

```
select * from super order by `row id` limit 6000
```

**Note:** we use the limit because while uploading the data only 1000 rows were uploaded for uploading the whole data we use the limit of 6000 so that we can upload the whole data we have cleaned.

- **Purpose:** This query retrieves all columns from the super table, sorted by the row id, and limits the result to 6000 rows.
- **Use Case:** Fetching a large dataset for analysis, focusing on specific columns that will be further processed.

### **(ii). Query: Count Customers by Payment Mode**

```
select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
```

The screenshot shows a SQL IDE interface. The query editor contains the following SQL statements:

```

1 • use da
2 select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9 select month(`order date`) from super
10

```

The result grid shows the output of the third query:

payment mode	no_of_customer
Online	2091
Cards	1245
COD	2371

- **Purpose:** This query counts the number of customers grouped by each payment mode.
- **Output:** Returns the payment mode with a corresponding count of customers.
- **Use Case:** Analyzing the popularity of different payment methods among customers.

### (iii). Query: Count Sales by Region

select(region),count(region) as sales from super group by region



The screenshot shows a SQL IDE interface. The query editor contains the following SQL statements:

```

1 • use da
2 ❌ select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9 select month(`order date`) from super
10

```

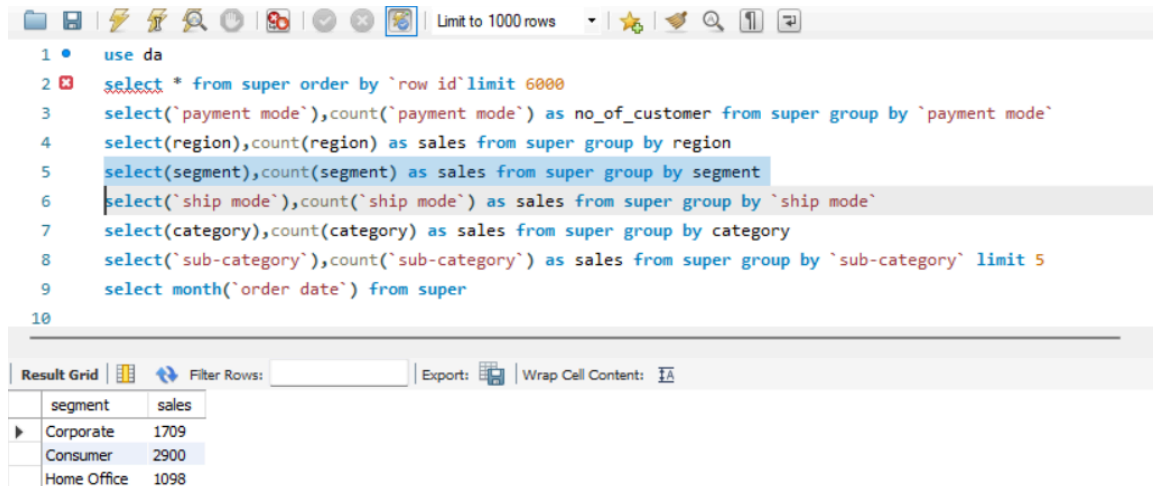
The result grid shows the output of the query executed on line 4:

region	sales
East	1632
West	1831
Central	1337
South	907

- **Purpose:** Count the number of sales in each region.
- **Output:** Displays the sales count for each region.
- **Use Case:** Useful for determining which geographical regions are performing best in terms of sales.

#### (iv). Query: Count Sales by Segment

select(segment),count(segment) as sales from super group by segment



```

1 • use da
2 select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9 select month(`order date`) from super
10

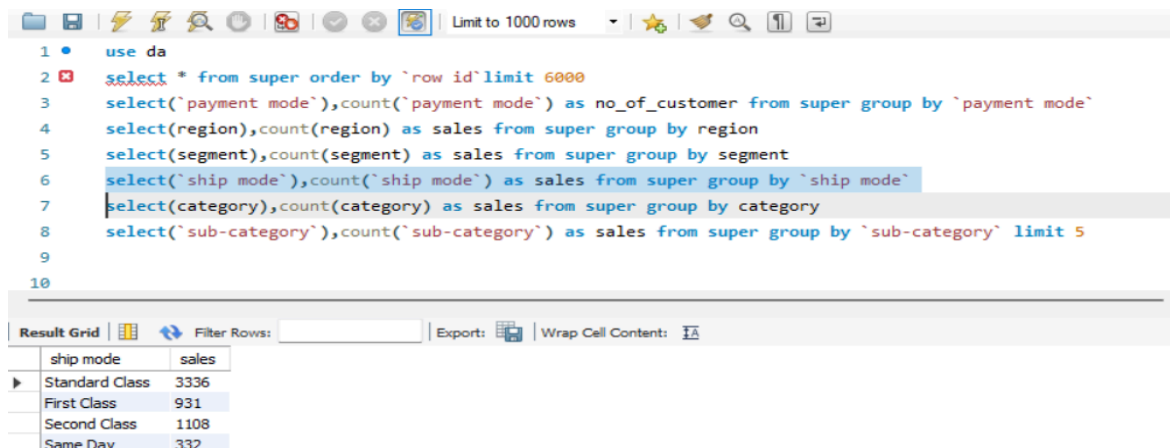
```

segment	sales
Corporate	1709
Consumer	2900
Home Office	1098

- **Purpose:** This counts sales in each customer segment (e.g., Consumer, Corporate, etc.).
- **Output:** Sales count by segment.
- **Use Case:** To see which customer segments contribute most to sales.

#### (v). Query: Count Sales by Shipping Mode

select(`ship mode`), count(`ship mode`) as sales from super group by `ship mode`



```

1 • use da
2 select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9
10

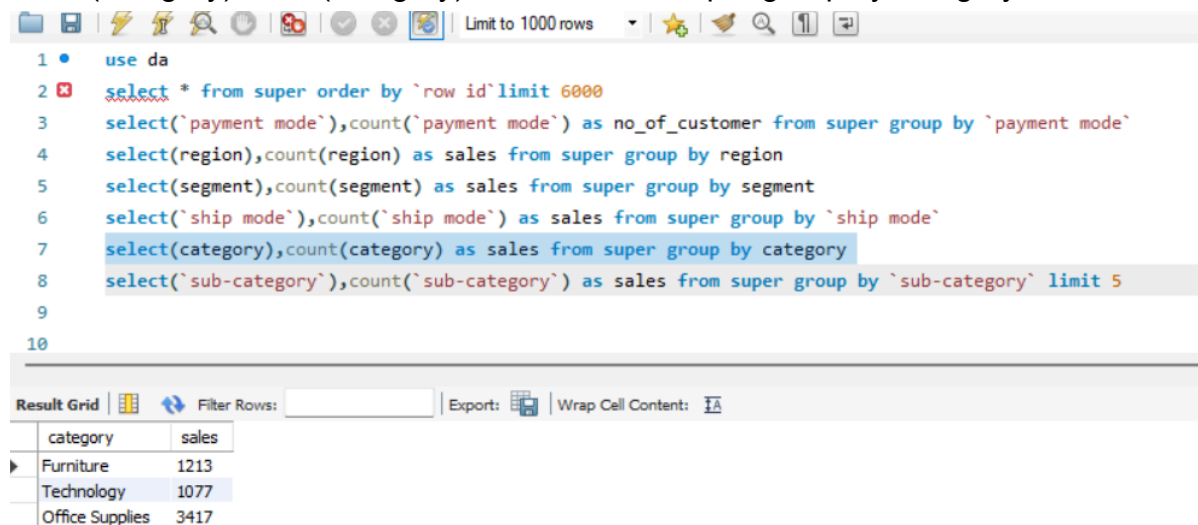
```

ship mode	sales
Standard Class	3336
First Class	931
Second Class	1108
Same Day	332

- **Purpose:** This query counts the sales grouped by different shipping modes.
- **Output:** Displays shipping modes and their corresponding sales count.
- **Use Case:** Helps to analyze how shipping preferences are distributed among customers.


#### (vi). Query: Count Sales by Category

select(category),count(category) as sales from super group by category



The screenshot shows a SQL IDE interface. The query editor at the top contains a series of SQL queries. The eighth query, which is highlighted, is: `select(category),count(category) as sales from super group by category`. Below the editor, the 'Result Grid' tab is active, displaying the results of the highlighted query. The grid has two columns: 'category' and 'sales'. It lists three categories: Furniture (1213), Technology (1077), and Office Supplies (3417).

```
1 • use da
2 ❌ select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9
10
```

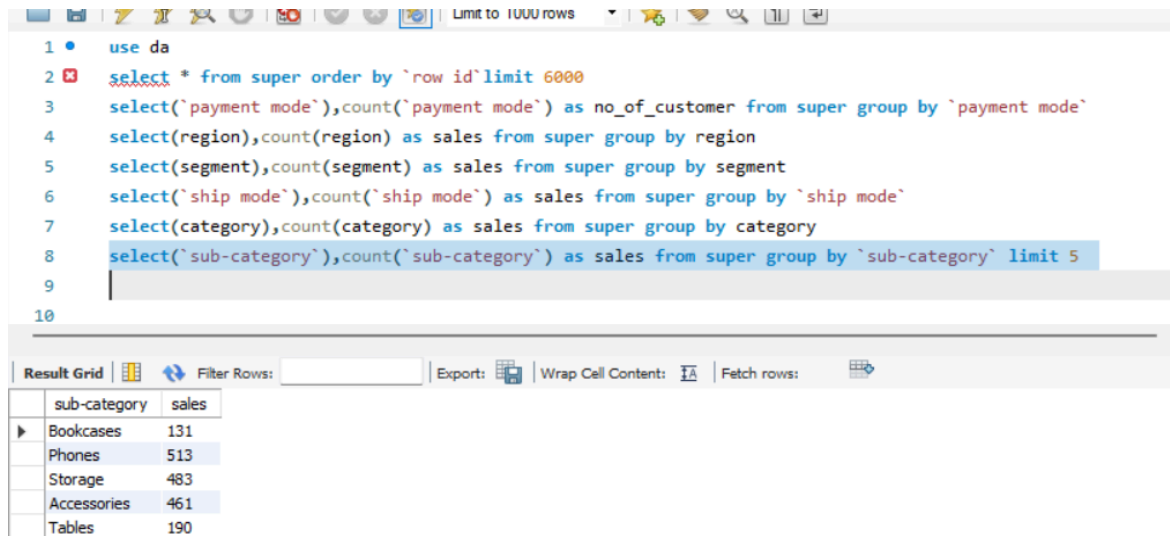
Result Grid | Filter Rows: | Export: | Wrap Cell Content: 

category	sales
Furniture	1213
Technology	1077
Office Supplies	3417

- **Purpose:** This counts the number of sales for each product category.
- **Output:** Sales count for each category.
- **Use Case:** To evaluate which product categories have the highest number of sales.

#### (vii). Query: Count Sales by Sub-Category (Limit 5)

select(`sub-category`), count(`sub-category`) as sales from super group by  
`sub-category` limit 5



The screenshot shows a SQL IDE interface. The top toolbar includes icons for file operations, a search icon, and a dropdown menu set to 'Limit to 1000 rows'. The query editor contains the following SQL code:

```
1 • use da
2 select * from super order by `row id` limit 6000
3 select(`payment mode`),count(`payment mode`) as no_of_customer from super group by `payment mode`
4 select(region),count(region) as sales from super group by region
5 select(segment),count(segment) as sales from super group by segment
6 select(`ship mode`),count(`ship mode`) as sales from super group by `ship mode`
7 select(category),count(category) as sales from super group by category
8 select(`sub-category`),count(`sub-category`) as sales from super group by `sub-category` limit 5
9
10
```

Below the editor is the 'Result Grid' section, which includes a 'Filter Rows' input field, an 'Export' button, a 'Wrap Cell Content' checkbox, and a 'Fetch rows' button. The result grid displays the following data:

sub-category	sales
Bookcases	131
Phones	513
Storage	483
Accessories	461
Tables	190

- **Purpose:** This counts the sales by sub-category and limits the result to 5.
- **Output:** Displays sales count for the top 5 sub-categories.
- **Use Case:** Useful for focusing on the top-performing sub-categories in the product line.

After analyzing the data, we further visualized it in Python as we couldn't visualize it in SQL.

### 3. Visualization of the Dataset(Python)

This code snippet demonstrates the importation of various essential libraries for data manipulation, visualization, and database interaction in Python. These libraries are

crucial for performing data analysis, creating visualizations, and connecting to databases.

## Libraries Imported

```
import mysql.connector  
import pandas as pd
```

```
import pandas as pd  
import seaborn as sns  
import matplotlib.pyplot as plt  
import numpy as np  
import plotly.express as px  
import plotly.graph_objects as go
```

### 1. MySQL. connector:

- Purpose: Used for connecting to MySQL databases.
- Usage: Facilitates database operations such as querying and updating records.

### 2. pandas (PD):

- Purpose: A powerful data manipulation and analysis tool.
- Usage: Provides data structures like DataFrames to handle and manipulate data efficiently.

### 3. seaborn (SNS):

- Purpose: A statistical data visualization library.
- Usage: Used for creating attractive and informative statistical graphics.

### 4. matplotlib.pyplot (plt):

- Purpose: A plotting library for creating static, interactive, and animated visualizations.
- Usage: Provides a MATLAB-like interface for plotting.

### 5. NumPy (np):

- Purpose: A fundamental package for scientific computing with Python.
- Usage: Supports large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

### 6. plotly. express (px):

- Purpose: A high-level interface for drawing attractive and informative statistical graphics.
- Usage: Simplifies the process of creating complex visualizations.

#### 7. **plotly.graph\_objects (go):**

- Purpose: Used for creating lower-level graph objects.
- Usage: Provides more control over the creation of complex visualizations.

### Why do we import these libraries?

We imported these libraries as they play a crucial role in analyzing and visualizing the data.

- **MySQL.connector:** This library is essential for establishing a connection to a MySQL database, allowing for data retrieval and manipulation directly from the database.
- **Pandas:** Imported pandas for emphasis, it is a cornerstone for data manipulation, enabling efficient handling of large datasets.
- **seaborn:** Enhances the capability of matplotlib by providing a high-level interface for drawing attractive statistical graphics.
- **Matplotlib:** A versatile library for creating a wide range of static, animated, and interactive plots.
- **NumPy:** Fundamental for numerical computations, providing support for arrays and matrices.
- **plotly. express** and **plotly.graph\_objects:** These libraries create interactive and complex visualizations, making data analysis more insightful.

Now, moving further to importing the data for My SQL we have to write the below syntax:

```
conn=mysql.connector.connect(host="localhost",user="root",password='12345',db='da')
```

```
query="select * from super"
```

```
df=pd.read_sql(query,conn)
df
```

This code demonstrates how to establish a connection to a MySQL database and retrieve data from a table using pandas and MySQL. connector. The retrieved data is stored in a data frame for further analysis.

### 1. Establishing a Connection:

- The connection to the MySQL database is made using MySQL. connector. connect function.
- Parameters:
  - host="localhost": Specifies the hostname where the MySQL server is running. In this case, it's the local machine.
  - user="root": The username used to connect to the database.
  - password='12345': The password associated with the user account.
  - db='da': The name of the database to connect to, in this case, "da".

## **2. Defining the SQL Query:**

- A SQL query is defined to select all data from the table named super

## **3. Executing the Query and Loading Data:**

- The query is executed, and the data is loaded into a pandas DataFrame using `pd.read_sql()`.
- Parameters:
  - query: The SQL query string to be executed.
  - conn: The MySQL connection object

## **4. Displaying the Data:**

- The DataFrame `df` is returned, which contains the data fetched from the super table.
- Simply typing `df` in a Jupyter Notebook will display the contents of the DataFrame.

C:\Users\yash pc\AppData\Local\Temp\ipykernel\_9676\2545783912.py:1: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy.  
df=pd.read\_sql(query,conn)

	Row id	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	...	Region	Product ID	Category	Sub-Category	Product Name	Sales
0	4918	CA-2019-160304	2019-01-01	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	...	East	FUR-BO-10004709	Furniture	Bookcases	Bush Westfield Collection Bookcases, Medium Ch...	74
1	4919	CA-2019-160304	2019-01-02	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	...	East	FUR-BO-10004709	Furniture	Bookcases	Bush Westfield Collection Bookcases, Medium Ch...	174
2	4920	CA-2019-160304	2019-01-02	2019-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	...	East	TEC-PH-10000455	Technology	Phones	GE 30522EE2	232

We have successfully uploaded the dataset in Python. Now, we have to check the info as we already cleaned and analyzed the dataset through Ms. Excel and My SQL still we have to check that data is properly uploaded as per the datatype we set in MySQL so we have to use the info function.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5707 entries, 0 to 5706
Data columns (total 21 columns):
#   Column          Non-Null Count  Dtype  
---  -
0   Row id          5707 non-null  int64  
1   Order ID        5707 non-null  object  
2   Order Date      5707 non-null  object  
3   Ship Date       5707 non-null  object  
4   Ship Mode       5707 non-null  object  
5   Customer ID     5707 non-null  object  
6   Customer Name   5707 non-null  object  
7   Segment         5707 non-null  object  
8   Country         5707 non-null  object  
9   City            5707 non-null  object  
10  State           5707 non-null  object  
11  Region          5707 non-null  object  
12  Product ID      5707 non-null  object  
13  Category        5707 non-null  object  
14  Sub-Category    5707 non-null  object  
15  Product Name    5707 non-null  object  
16  Sales           5707 non-null  int64  
17  Quantity        5707 non-null  int64  
18  Profit          5707 non-null  int64  
19  Returns         5707 non-null  int64  
20  Payment Mode    5707 non-null  object  
dtypes: int64(5), object(16)
memory usage: 936.4+ KB
```



Everything is fine in our dataset only the issue arises in the column order date and ship date. It is in object format but it has to be in date and time format. so we have to change the data type of order date and ship date for that we have written the below syntax:

```
from datetime import date
df2=df.astype({'Order Date':'datetime64[ns]','Ship Date':'datetime64[ns]'})
```

- **Importing Libraries:** The date-time module is imported to handle date operations.
- **Data Conversion:** The as-type method is used to convert the 'Order Date' and 'Ship Date' columns to DateTime.

Now, again we have to use the info function to check whether changes are visible or not.

```
df2.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5707 entries, 0 to 5706
Data columns (total 21 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   Row id          5707 non-null   int64
1   Order ID        5707 non-null   object
2   Order Date      5707 non-null   datetime64[ns]
3   Ship Date       5707 non-null   datetime64[ns]
4   Ship Mode       5707 non-null   object
5   Customer ID     5707 non-null   object
6   Customer Name   5707 non-null   object
7   Segment        5707 non-null   object
8   Country         5707 non-null   object
9   City           5707 non-null   object
10  State           5707 non-null   object
11  Region          5707 non-null   object
12  Product ID      5707 non-null   object
13  Category        5707 non-null   object
14  Sub-Category    5707 non-null   object
15  Product Name    5707 non-null   object
16  Sales           5707 non-null   int64
17  Quantity        5707 non-null   int64
18  Profit          5707 non-null   int64
19  Returns         5707 non-null   int64
20  Payment Mode    5707 non-null   object
dtypes: datetime64[ns](2), int64(5), object(14)
memory usage: 936.4+ KB
```

As per the above image, we know the changes we made are reflected in the dataset.

```
df2.describe()
```

	Row id	Order Date	Ship Date	Sales	Quantity	Profit	Returns
count	5707.000000	5707	5707	5707.000000	5707.000000	5707.000000	5707.000000
mean	5026.035045	2020-02-28 15:04:04.121254656	2020-03-03 13:17:20.196250112	269.182057	3.778167	30.299807	0.047836
min	1.000000	2019-01-01 00:00:00	2019-01-05 00:00:00	1.000000	1.000000	-6600.000000	0.000000
25%	2499.500000	2019-09-11 00:00:00	2019-09-14 00:00:00	72.500000	2.000000	2.000000	0.000000
50%	5114.000000	2020-03-16 00:00:00	2020-03-18 00:00:00	129.000000	3.000000	9.000000	0.000000
75%	7455.500000	2020-09-12 00:00:00	2020-09-15 00:00:00	270.000000	5.000000	29.000000	0.000000
max	9994.000000	2020-12-31 00:00:00	2021-01-05 00:00:00	9100.000000	14.000000	8400.000000	1.000000
std	2878.126367	NaN	NaN	480.859947	2.212239	263.807847	0.213438

We have used the describe function to analyze the numerical terms like mean , std

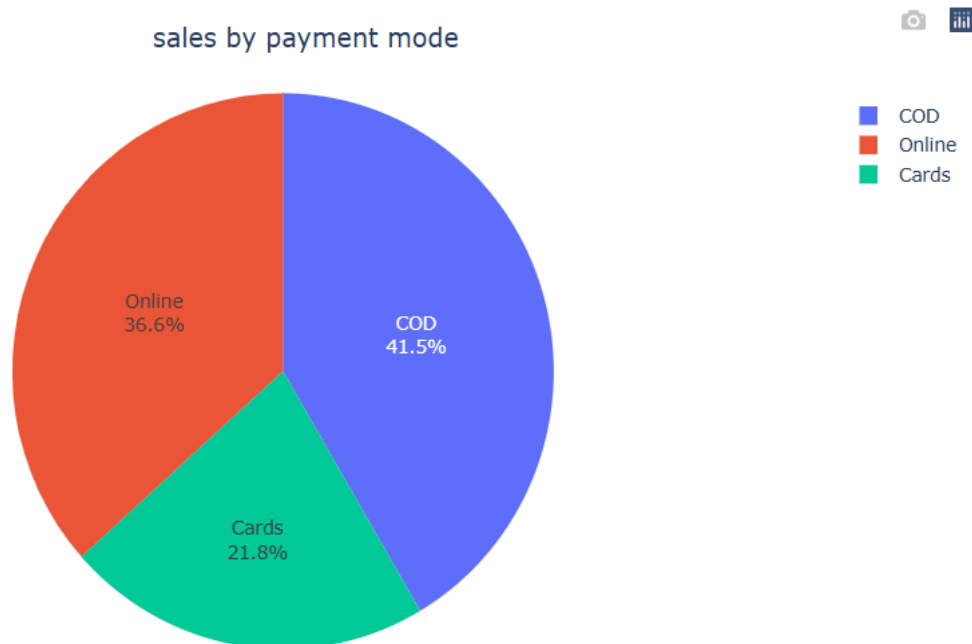
Now, we have to visualize the dataset as we have already analyzed.

## 1. Sales by Payment Mode

```
paymentmode=df.groupby(df["Payment Mode"],as_index=False)["Sales"].count()
paymentmode
```

	Payment Mode	Sales
0	COD	2371
1	Cards	1245
2	Online	2091

```
fig = px.pie(paymentmode, values='Sales', names='Payment Mode',width=900,height=500)
fig.update_layout(title={'text':'sales by payment mode','x':0.5,'xanchor':"center"})
fig.update_traces(textposition='inside', textinfo='percent+label')
fig.show()
```



For generating a pie chart visualization of sales by payment mode using Python, pandas, and Plotly Express. The chart illustrates sales distribution across three payment modes: Cash on Delivery (COD), Online, and Cards.

Now, we have to group the data by the Payment Mode column and count the number of sales for each mode using `payment_mode = df.groupby('Payment Mode')['Sales'].count()`. Create a pie chart with plotly express by executing `fig = px.pie(payment_mode, values='Sales', names='Payment Mode', title='Sales by Payment Mode', width=500, height=500)`. Enhance the chart by updating the traces to display both percentage and label information with `fig.update_traces(textinfo='percent+label')`. Finally, display the chart using `fig.show()`.

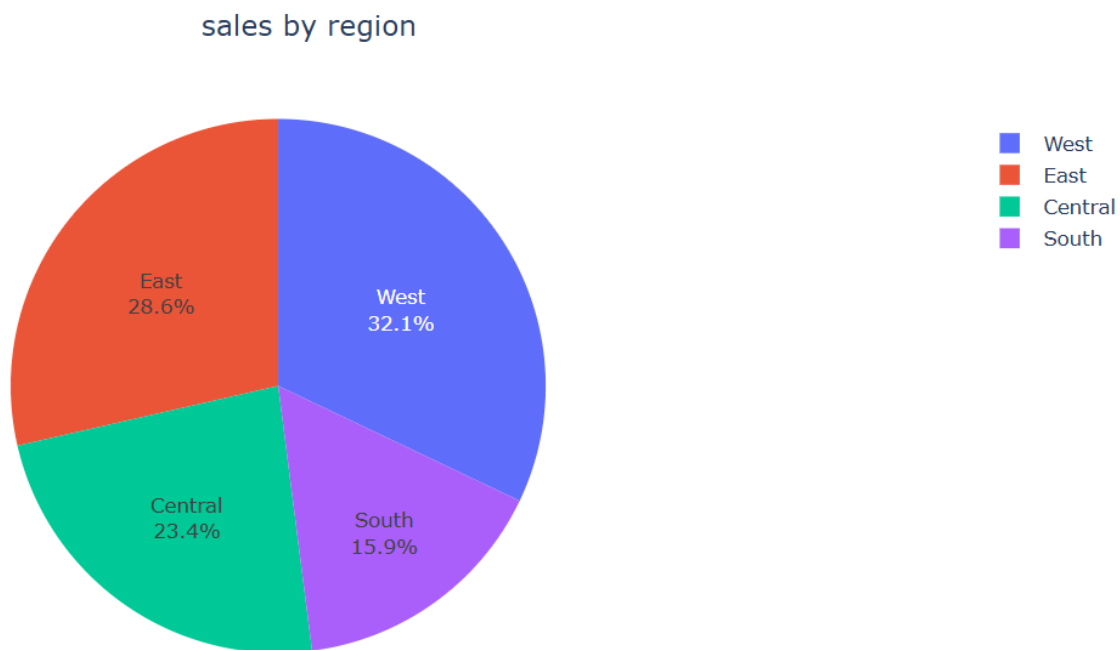
This visualization effectively communicates the sales distribution across different payment modes, providing valuable insights into customer preferences.

## 2. Sales by Region

```
region=df.groupby(df["Region"],as_index=False)["Sales"].count()  
region
```

	Region	Sales
0	Central	1337
1	East	1632
2	South	907
3	West	1831

```
# plt.title("Sale by Region",loc="center")  
fig = px.pie(region, values='Sales', names='Region',title="sales by region",width=1000,height=500)  
fig.update_layout(title={'text':'sales by region','x':0.5,'xanchor':"center"})  
  
fig.update_traces(textposition='inside', textinfo='percent+label')  
fig.show()
```



It calculates the total sales for each region and the percentage of total sales. The code then generates a pie chart to represent the sales distribution across the regions visually. The pie chart is labeled with the region names and the corresponding sales percentages, ensuring an equal aspect ratio for a circular pie chart. The chart is titled

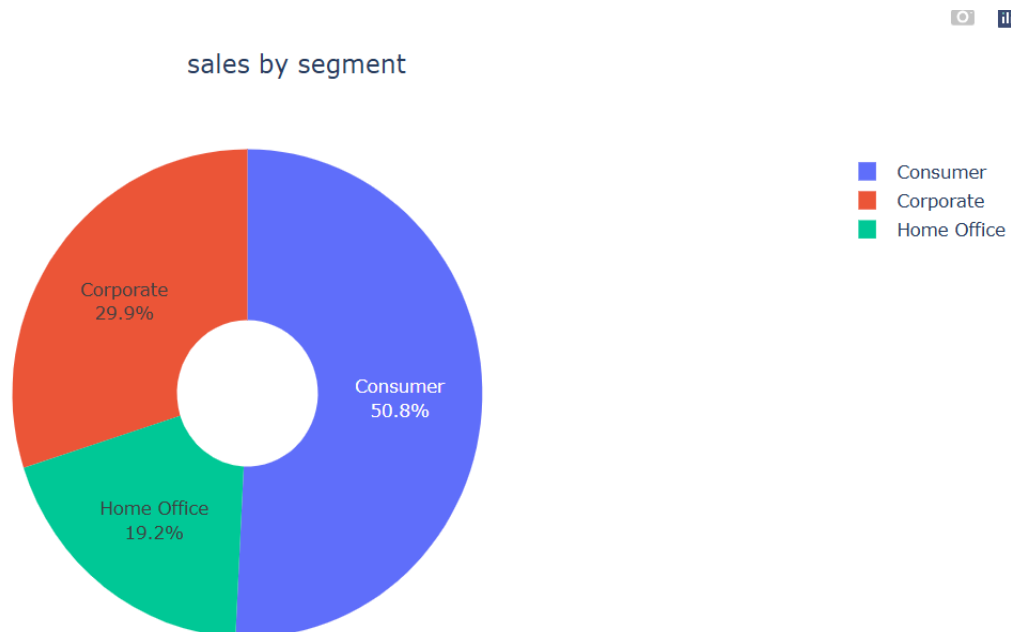
“Sales by Region” and formatted with a specific font size for clarity. The regions included are Central, East, South, and West, with their respective sales and percentages displayed in the chart.

### 3. Sales by Segment

```
segment=df.groupby(df["Segment"],as_index=False)["Sales"].count()  
segment
```

	Segment	Sales
0	Consumer	2900
1	Corporate	1709
2	Home Office	1098

```
fig = px.pie(segment, values='Sales', names='Segment',title='sales by segment',width=1000,height=500,hole=.3)  
fig.update_layout(title={'text':'sales by segment','x':0.5,'xanchor':"center"})  
  
fig.update_traces(textposition='inside', textinfo='percent+label')  
fig.show()
```



This script generates a pie chart to visualize the distribution of sales across different segments (Consumer, Corporate, and Home Office). The chart helps in understanding the proportion of sales contributed by each segment, which can guide business decisions.

### a.Data Grouping by Segment:

```
segment = df.groupby(df["Segment"], as_index=False)["Sales"].count()
```

- **Purpose:** Groups the sales data based on the "Segment" column and counts the number of sales for each segment.
- **Result:** A data frame is created, summarizing the number of sales in each segment (Consumer, Corporate, and Home Office).

### b.Pie Chart Creation:

```
fig = px.pie(segment, values='Sales', names='Segment', title='sales by segment', width=1000, height=500, hole=.3)
```

- **Purpose:** Create a pie chart using Plotly Express.
- **Parameters:**
  - `values='Sales'`: Specifies the data column containing the sales count to plot.
  - `names='Segment'`: Labels each slice of the pie chart with the corresponding segment name.
  - `title='sales by segment'`: Sets the title of the pie chart.
  - `width` and `height`: Customize the size of the chart.
  - `hole=.3`: Defines the size of the hole in the center, making it a donut chart.

### c.Updating Layout:

```
fig.update_layout(title={'text':'sales by segment','x':0.5,'xanchor':'center'})
```

- **Purpose:** Customize the chart's layout by centering the title.

### d.Trace Updates (Adding Percentages and Labels Inside the Chart):

```
fig.update_traces(textposition='inside', textinfo='percent+label')
```

- **Purpose:** Ensures that the percentage and segment labels appear inside each pie slice for better readability.

### e.Displaying the Chart:

```
fig.show()
```

- **Purpose:** Displays the final pie chart in the output.

The pie chart displays the following distribution:

- **Consumer Segment:** 50.8% of total sales.

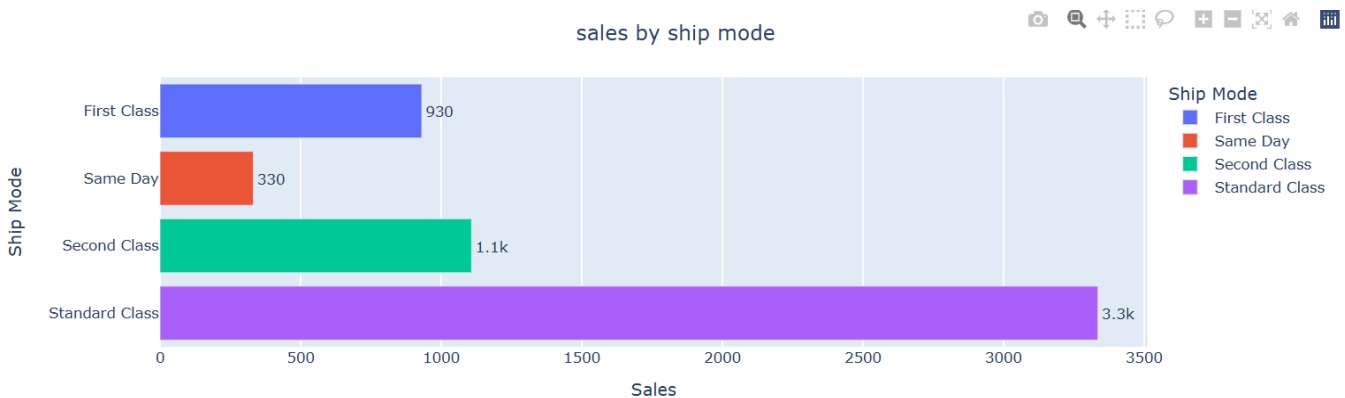
- **Corporate Segment:** 29.9% of total sales.
- **Home Office Segment:** 19.2% of total sales.

## 4. Sales by ShipMode

```
shipmode=df.groupby(df["Ship Mode"],as_index=False)["Sales"].count()
shipmode
```

	Ship Mode	Sales
0	First Class	931
1	Same Day	332
2	Second Class	1108
3	Standard Class	3336

```
fig=px.bar(shipmode,y="Ship Mode",x="Sales",labels=shipmode["Ship Mode"],color=shipmode["Ship Mode"],text_auto='.2s',orientation='h')
fig.update_layout(title={'text':'sales by ship mode','x':0.5,'xanchor':'center'})
fig.update_traces(textfont_size=12,textangle=0,textposition="outside",cliponaxis=False) # This line is used to make the value outside the bar.
fig.show()
```



The pandas library in Python to analyze sales data by ship mode. The code begins by grouping the DataFrame `df` by the "Ship Mode" column using the group by method. The `as_index=False` parameter ensures that the grouped column is not set as the index of the resulting DataFrame. The `count()` method is then applied to count the occurrences of each unique value in the "Ship Mode" column. The resulting DataFrame, stored in the variable `shipmode`, contains two columns: "Ship Mode" and "Sales". The output table shows the counts of sales for each shipping mode: 'First Class' with 931 sales, 'Same Day' with 332 sales, 'Second Class' with 1108 sales, and 'Standard Class' with 3336 sales. This analysis helps in understanding the distribution of sales across different shipping modes, providing valuable insights for decision-making in logistics and supply chain management.

The Plotly Express library to create a horizontal bar chart visualizing sales data by ship mode. The `px.bar` function is used to generate the bar chart, with the `y` parameter set to “Ship Mode” and the `x` parameter set to “Sales”. The `labels` parameter customizes the labels for the ship modes, and the `color` parameter assigns different colors to each ship mode. The `text_auto` parameter formats the text displayed on the bars to two decimal places, and the `orientation` parameter specifies a horizontal orientation for the bars.

The layout of the chart is updated using the `fig.update_layout` method, which sets the title of the chart to “sales by ship mode” and centers it horizontally. The `fig.update_traces` method is then used to customize the appearance of the text on the bars, setting the font size to 12, the text angle to 0, the text position to “outside”, and ensuring that the text is not clipped by the axis. Finally, the `fig.show()` method is called to display the chart. This code effectively creates a visually appealing and informative bar chart that represents sales data by ship mode.

## 5. Sales by Category

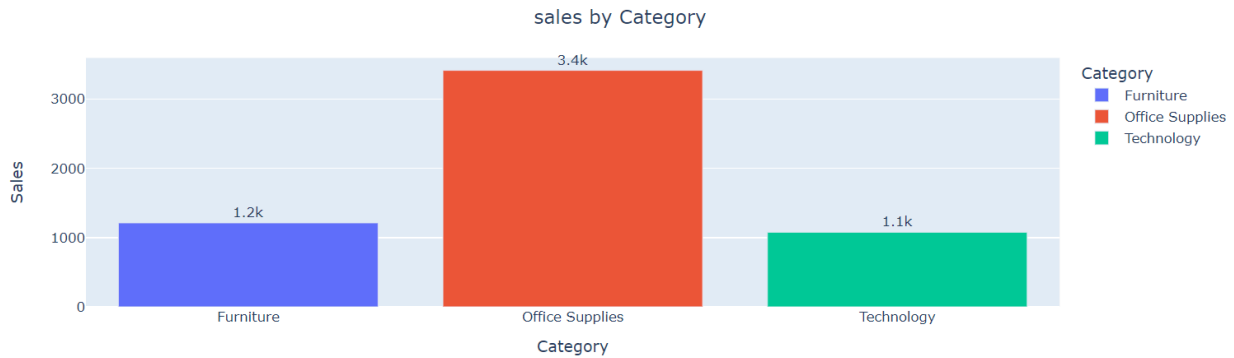
```
category=df.groupby(df["Category"],as_index=False)["Sales"].count()  
category
```

	Category	Sales
0	Furniture	1213
1	Office Supplies	3417
2	Technology	1077

The `pandas` library in Python analyzes sales data by category. The code begins by grouping the `DataFrame` `df` by the “Category” column using the `group by` method. The `as_index=False` parameter ensures that the grouped column is not set as the index of the resulting `DataFrame`. The `count()` method is then applied to count the occurrences of each unique value in the “Sales” column. The resulting `DataFrame`, stored in the variable `category`, contains two columns: “Category” and “Sales”. The output table shows the counts of sales for each category: ‘Furniture’ with 1213 sales, ‘Office Supplies’ with 3417 sales, and ‘Technology’ with 1077 sales. This analysis helps in understanding the distribution of sales across different categories, providing valuable insights for decision-making in inventory management and sales strategy.



```
fig=px.bar(category,x="Category",y="Sales",labels=category["Category"],color=category["Category"],text_auto='.2s')
fig.update_layout(title={'text':'sales by ship mode','x':0.5,'xanchor':'center'})
fig.update_traces(textfont_size=12,textangle=0,textposition="outside",cliponaxis=False)
fig.show()
```



The code begins by defining a bar chart with the `px. bar` function, specifying the x-axis as “Category” and the y-axis as “Sales”. The `labels` parameter is used to label the categories, and the `color` parameter assigns different colors to each category. The `text_auto='.2s'` parameter ensures that the text on the bars is formatted to two significant figures.

Next, the `fig.update_layout` function is used to customize the layout of the chart. The title of the chart is set to ‘Sales by ship mode’, and it is centered using the `x` and `xanchor` parameters. The `fig.update_traces` function further customizes the appearance of the text on the bars, setting the font size to 12, the text angle to 0, the text position to “outside”, and ensuring that the text is not clipped by the axis.

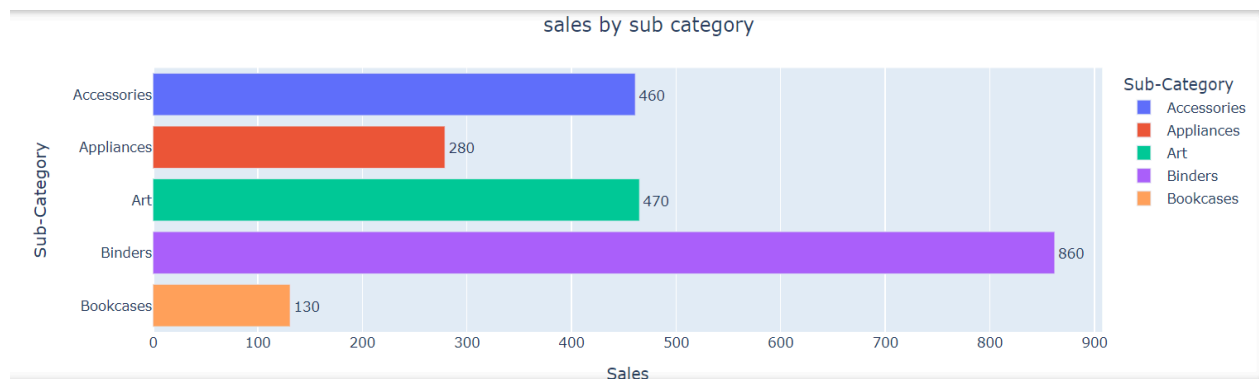
Finally, the `fig. show()` function is called to display the chart. This code snippet provides a clear example of how to use Plotly Express to create a visually appealing and informative bar chart, with various customization options to enhance the clarity and presentation of the data.

## 6. Sales by Sub-category

```
subcategory=df.groupby(df["Sub-Category"],as_index=False)["Sales"].count().head()  
subcategory
```

	Sub-Category	Sales
0	Accessories	461
1	Appliances	279
2	Art	465
3	Binders	862
4	Bookcases	131

```
fig=px.bar(subcategory,y="Sub-Category",x="Sales",labels=subcategory["Sub-Category"],color=subcategory["Sub-Category"],text_auto='.2s')  
fig.update_layout(title={'text':'sales by sub category','x':0.5,'xanchor':'center'})  
fig.update_traces(textfont_size=12,textangle=0,textposition="outside",cliponaxis=False)  
fig.show()
```



The pandas library performs a group-by operation on a DataFrame. The code begins by importing the pandas library, which is essential for data manipulation and analysis in Python. A sample data frame is created with two columns: 'Sub-Category' and 'Sales'. This data frame includes various sub-categories such as Accessories, Appliances, Art, Binders, and Bookcases, along with their respective sales figures. The core of the operation involves grouping the data by the 'Sub-Category' column using the group by method. This method aggregates the data, allowing for a count of occurrences in the 'Sales' column for each sub-category. The result is a new data frame that displays the sub-categories and their corresponding sales counts. The .head() function is used to display the top five results, providing a quick overview of the most frequent sub-categories.

## 7. Sales by Month and Year

```
sales2019=df2[pd.DatetimeIndex(df2['Order Date']).year==2019]
sales2020=df2[pd.DatetimeIndex(df2['Order Date']).year==2020]
```

To filter sales data for specific years using pandas in Python, you can use the following approach. First, ensure that your sales data is in a pandas DataFrame and that the 'Order Date' column is in DateTime format. Then, you can filter the data for the years 2019 and 2020.

```
sales2019year=pd.DatetimeIndex(sales2019['Order Date']).year
sales2019month=pd.DatetimeIndex(sales2019['Order Date']).month
```

To extract the year and month from the 'Order Date' column in the sales2019 DataFrame, you can use the pd.DatetimeIndex method from the pandas library. First, create a variable named sales2019Year and assign it the result of calling pd.DatetimeIndex on sales2019['Order Date'], then access the .year attribute. Similarly, create a variable named sales2019Month and assign it the result of calling pd.DatetimeIndex on sales2019['Order Date'], then access the .month attribute. This allows you to efficiently extract and work with the year and month components of the date field for further analysis or manipulation.

```
sales2019copy=sales2019.copy()
sales2020copy=sales2020.copy()
```

To create copies of data structures in Python, you can use the .copy() method from the pandas library. This method is essential for data manipulation as it allows you to work with a duplicate of the original data without altering it. For instance, if you have two DataFrames, sales2019, and sales2020, you can create their copies by using the above syntax.

This ensures that any modifications made to sales2019copy or sales2020copy do not affect the original sales2019 and sales2020 DataFrames. This practice is particularly useful when you need to perform operations that might change the data, allowing you to preserve the original dataset for reference or further use.

```
sales2019copy["Order Date"]=sales2019month
```

the pandas library in Python to perform data manipulation. Specifically, it groups the sales2019 DataFrame by the 'Order Date' column and then aggregates the 'Sales' column by summing its values. The code attempts to access the .month attribute from the result, which suggests further processing based on the month of the order dates.

```
month2019sales=sales2019copy.groupby(sales2019copy["Order Date"],as_index=False)["Sales"].sum()  
month2019sales
```

	Order Date	Sales
0	1	18512
1	2	19707
2	3	50832
3	4	38594
4	5	50462
5	6	39637
6	7	39108
7	8	31017
8	9	71853
9	10	41125
10	11	78457
11	12	77484

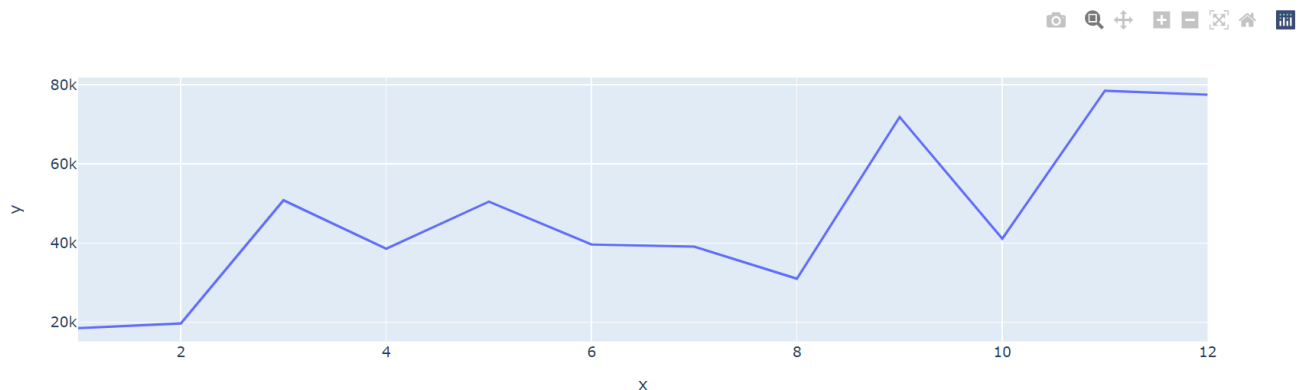
The pandas library for data manipulation and analysis. The code snippet demonstrates how to group data by the 'Order Date' column in a DataFrame named 'sales2019', and then calculate the sum of the 'Sales' for each group. The resulting DataFrame, 'month 2019 sales', contains two columns: 'Order Date' and 'Sales'. The 'Order Date' column represents aggregated periods, likely months, while the 'Sales' column displays the total sales figures for each period.

```
: month2020sales=sales2020copy.groupby(sales2020copy["Order Date"],as_index=False)["Sales"].sum()
month2020sales
```

	Order Date	Sales
0	1	54261
1	2	51011
2	3	63487
3	4	50080
4	5	65902
5	6	67588
6	7	58661
7	8	76423
8	9	117141
9	10	87653
10	11	128844
11	12	158383

Initially, the sales 2020 copy DataFrame is grouped by the “Order Date” column using the group by method. The as\_index=False parameter ensures that the “Order Date” remains a column in the resulting DataFrame rather than becoming the index. The sum() function is then applied to aggregate the sales data for each order date. The final result is stored in the month 2020 sales DataFrame, which contains two columns: “Order Date” and “Sales,” displaying the total sales for each date. This approach efficiently summarizes sales data, making it easier to analyze trends over time.

```
px.line(x=month2020sales["Order Date"],y=month2020sales["Sales"])
px.line(x=month2019sales["Order Date"],y=month2019sales["Sales"])
```



The line graph represents monthly sales trends over a year. The horizontal axis, labeled ‘x’, ranges from 1 to 12, likely representing months, while the vertical axis, labeled ‘y’,

ranges from 0 to 800K, presumably indicating sales figures in thousands. The graph illustrates fluctuations in sales, with notable peaks around the fifth and last months.

The graph is generated using Python's Plotly library, as indicated by the code snippet:

```
px.line(x=monthly2019Sales['Order Date'], y=monthly2019Sales['Sales']).
```

```
px.line(x=monthly2020Sales['Order Date'], y=monthly2020Sales['Sales']).
```

This suggests that the data is sourced from a dataset named monthly2019Sales and monthly2020Sales with columns 'Order Date' and 'Sales'.

The sales trends over the year depict that there are significant increases in sales around the fifth and last months. This data is visualized using Python's Plotly library, highlighting the fluctuations in monthly sales.

This shows the sales of both years online so we need the new line graph with two different lines representing sales of two different years.

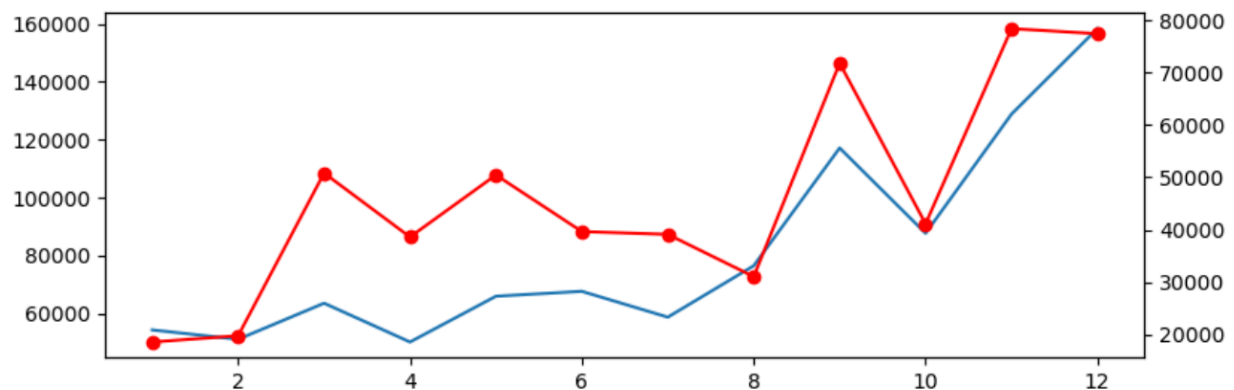
```
plt.figure(figsize=(9, 3))
```

```
axes1 = plt.subplot()
```

```
b = axes1.plot(month2020sales["Order Date"], month2020sales["Sales"])
```

```
axes2 = axes1.twinx()
```

```
p = axes2.plot(month2019sales["Order Date"], month2019sales["Sales"], c='r', marker='o')
```



The attached image shows a line graph with two y-axes plotted using a Python library, likely Matplotlib. On the left y-axis, values range from 0 to 160,000, and on the right y-axis, values range from 0 to 80,000. The x-axis represents months, numbered from 1 to 12. There are two lines: one in blue and one in red with markers. The blue line corresponds to the left y-axis and shows a fluctuating trend with peaks and troughs. The red line corresponds to the right y-axis and also shows fluctuations but with an overall increasing trend towards the end of the year.

The graph is generated using Python's matplotlib library, as indicated by the code snippet:

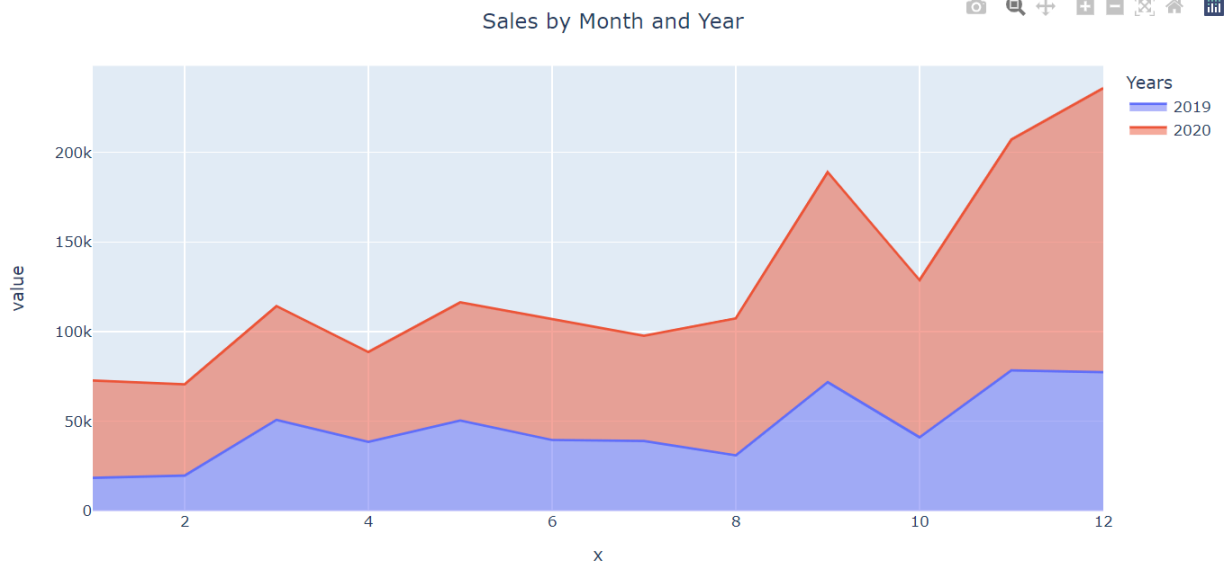
```
plt.figure(figsize=(9, 3)),  
  
axes1 = plt.subplot(),  
  
b = axes1.plot(month2020sales["Order Date"], month2020sales["Sales"]),  
  
axes2 = axes1.twinx(),  
  
p = axes2.plot(month2019sales["Order Date"], month2019sales["Sales"], c='r',  
marker='o').
```

This suggests that the data is sourced from datasets named month2020sales and month2019sales with columns 'Order Date' and 'Sales'.

The sales trends over the year depict that there are significant fluctuations in sales, with notable peaks and troughs in the blue line representing 2020 sales and an overall increasing trend in the red line representing 2019 sales towards the end of the year. This data is visualized using Python's matplotlib library, highlighting the differences in sales trends between the two years.

Now, We are making an area chart to make it more presentable.

```
fig = px.area(x=month2019sales["Order Date"],y=[month2019sales["Sales"],month2020sales["Sales"]],width=1000,height=500)  
fig.update_layout(title={'text':'Sales by Month and Year','x':0.5,'xanchor':'center'})  
fig.update_layout(legend={"title":"Years"})  
  
newnames = {'wide_variable_0':'2019', 'wide_variable_1': '2020'}  
fig.for_each_trace(lambda t: t.update(name = newnames[t.name],  
                                     legendgroup = newnames[t.name],  
                                     hovertemplate = t.hovertemplate.replace(t.name, newnames[t.name])  
                                     )  
                )  
  
fig.show()
```



The attached image displays a line graph with a shaded area, representing sales data over various months for the years 2019 and 2020. The x-axis indicates the month, numbered from 1 to 12, while the y-axis shows sales figures in thousands of units, ranging from 0 to approximately 250k. Two lines are plotted: one for each year, with the area below each line filled with color—blue for 2019 and red for 2020. This graph visually compares monthly sales figures across two consecutive years, highlighting trends or changes in sales performance over time.

## 8. Profit by Month and Year

```
profit2019=df2[pd.DatetimeIndex(df2['Order Date']).year==2019]  
profit2020=df2[pd.DatetimeIndex(df2['Order Date']).year==2020]
```

The code in the image is used to filter a data frame named `df2` based on the year of the "Order Date" column. The result is stored in two new DataFrames, `profit2019` and `profit2020`, respectively.

Here's a breakdown of the code:

### Line 1:

```
profit2019 = df2[pd.DatetimeIndex(df2['Order Date']).year == 2019]
```

- `df2`: This is the original data frame containing the data.



- `pd.DatetimeIndex(df2['Order Date'])`: This line converts the "Order Date" column to a datetime index, which is necessary for filtering based on the year.
- `.year == 2019`: This condition filters the rows where the year of the "Order Date" is 2019. The resulting filtered DataFrame is stored in `profit2019`.

#### Line 2:

```
profit2020 = df2[pd.DatetimeIndex(df2['Order Date']).year == 2020]
```

- This line is similar to the first line, but it filters the rows where the year of the "Order Date" is 2020. The resulting filtered DataFrame is stored in `profit2020`

```
profit2019year=pd.DatetimeIndex(profit2019['Order Date']).year
profit2019month=pd.DatetimeIndex(profit2019['Order Date']).month
```

The code in the image extracts the year and month from the "Order Date" column of the `profit2019` DataFrame.

Here's a breakdown of the code:

#### Line 1:

```
profit2019year = pd.DatetimeIndex(profit2019['Order Date']).year
```

- `profit2019`: This is the data frame containing the data for the year 2019.
- `pd.DatetimeIndex(profit2019['Order Date'])`: This line converts the "Order Date" column to a datetime index, which is necessary for extracting the year.
- `.year`: This extracts the year component from the datetime index and stores it in the `profit2019year` variable.

#### Line 2:

```
profit2019month = pd.DatetimeIndex(profit2019['Order Date']).month
```

- This line is similar to the first line, but it extracts the month component from the datetime index and stores it in the `profit2019month` variable.

```
profit2019copy=profit2019.copy()
profit2020copy=profit2020.copy()
```

The above code creates copies of the profit2019 and profit2020 DataFrames.

Here's a breakdown of the code:

**Line 1:**

```
profit2019copy = profit2019.copy()
```

- profit2019: This is the original data frame containing the data for the year 2019.
- .copy(): This method creates a deep copy of the profit2019 DataFrame. A deep copy creates a new object in memory, completely independent of the original object. Any changes made to the profit2019copy DataFrame will not affect the original profit2019 DataFrame.
- profit2019copy: This is the new variable that stores the copy of the profit2019 DataFrame.

**Line 2:**

```
profit2020copy = profit2020.copy()
```

- This line is similar to the first line, but it creates a copy of the profit2020 DataFrame and stores it in the profit2020copy variable.

```
profit2020copy=profit2020.copy()
```

The code in the image creates a copy of the profit2020 DataFrame.

Here's a breakdown of the code:

**Line 1:**

```
profit2020copy = profit2020.copy()
```

- profit2020: This is the original data frame containing the data for the year 2020.
- .copy(): This method creates a deep copy of the profit2020 DataFrame. A deep copy creates a new object in memory, completely independent of the original object. Any changes made to the profit2020copy DataFrame will not affect the original profit2020 DataFrame.
- profit2020copy: This is the new variable that stores the copy of the profit2020 DataFrame.

```
profit2019copy["Order Date"]=profit2019month
```

The code in the image assigns the values from the profit2019month Series to the "Order Date" column of the profit2019copy DataFrame.

Here's a breakdown of the code:

**Line 1:**

```
profit2019copy["Order Date"] = profit2019month
```

- profit2019copy: This is the data frame where the changes will be made.
- "Order Date": This specifies the column in the profit2019copy DataFrame that will be modified.
- = profit2019month: This assigns the values from the profit2019month Series to the specified column. The profit2019month Series should have the same length as the number of rows in the profit2019copy DataFrame.

```
profit2020year=pd.DatetimeIndex(profit2020['Order Date']).year  
profit2020month=pd.DatetimeIndex(profit2020['Order Date']).month
```

The code in the image extracts the year and month from the "Order Date" column of the profit2020 DataFrame.

Here's a breakdown of the code:

**Line 1:**

```
profit2020year = pd.DatetimeIndex(profit2020['Order Date']).year
```

- profit2020: This is the data frame containing the data for the year 2020.
- pd.DatetimeIndex(profit2020['Order Date']): This line converts the "Order Date" column to a datetime index, which is necessary for extracting the year.
- .year: This extracts the year component from the datetime index and stores it in the profit2020year variable.

**Line 2:**

```
profit2020month = pd.DatetimeIndex(profit2020['Order Date']).month
```

- This line is similar to the first line, but it extracts the month component from the datetime index and stores it in the profit2020month variable.

```
profit2020copy["Order Date"]=profit2020month
```

The code in the image assigns the values from the profit2020month Series to the "Order Date" column of the profit2020copy DataFrame.

Here's a breakdown of the code:

**Line 1:**

```
profit2020copy["Order Date"] = profit2020month
```

- profit2020copy: This is the data frame where the changes will be made.
- "Order Date": This specifies the column in the profit2020copy DataFrame that will be modified.
- = profit2020month: This assigns the values from the profit2020month Series to the specified column. The profit2020month Series should have the same length as the number of rows in the profit2020copy DataFrame.

```
month2019=profit2019copy.groupby(profit2019copy["Order Date"],as_index=False)["Profit"].sum()  
month2019
```

The code in the image calculates the sum of the "Profit" column for each month in the profit2019copy DataFrame.

Here's a breakdown of the code:

**Line 1:**

```
month2019 = profit2019 copy. group by(profit2019copy["Order Date"],  
as_index=False)["Profit"].sum()
```

- profit2019copy: This is the data frame containing the data for the year 2019.
- .groupby(profit2019copy["Order Date"], as\_index=False): This groups the DataFrame by the "Order Date" column, preserving the original index as a separate column.
- ["Profit"]: This selects the "Profit" column from the grouped DataFrame.
- .sum(): This calculates the sum of the "Profit" column for each group (i.e., for each month).

**Line 2:**

```
month2019
```

- This line simply displays the resulting DataFrame month of 2019.

```
month2020=profit2020copy.groupby(profit2020copy["Order Date"],as_index=False)["Profit"].sum()
month2020
```

The code in the image calculates the sum of the "Profit" column for each month in the profit2020copy DataFrame.

Here's a breakdown of the code:

#### Line 1:

```
month2020 = profit2020copy.groupby(profit2020copy["Order Date"],
as_index=False)["Profit"].sum()
```

- profit2020copy: This is the data frame containing the data for the year 2020.
- .groupby(profit2020copy["Order Date"], as\_index=False): This groups the DataFrame by the "Order Date" column, preserving the original index as a separate column.
- ["Profit"]: This selects the "Profit" column from the grouped DataFrame.
- .sum(): This calculates the sum of the "Profit" column for each group (i.e., for each month).

#### Line 2:

```
month2020
```

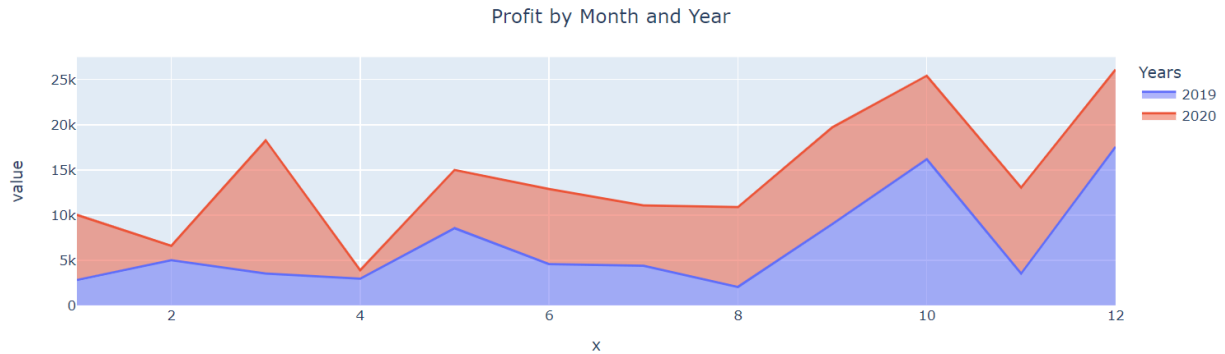
- This line simply displays the resulting DataFrame month 2020.

```
fig = px.area(x=month2019["Order Date"], y=[month2019["Profit"],month2020["Profit"]])
fig.update_layout(title={'text':'Profit by Month and Year','x':0.5,'xanchor':'center'})

fig.update_layout(legend={"title":"Years"})

newnames = {'wide_variable_0':'2019', 'wide_variable_1': '2020'}
fig.for_each_trace(lambda t: t.update(name = newnames[t.name],
                                     legendgroup = newnames[t.name],
                                     hovertemplate = t.hovertemplate.replace(t.name, newnames[t.name])
                                     )
)

fig.show()
```



The code in the image creates an area chart to visualize the profit trend over time for two years, 2019 and 2020.

Here's a breakdown of the code:

### Creating the Plot:

```
fig = px.area(x=month2019["Order Date"], y=[month2019["Profit"], month2020["Profit"]])
```

This line creates an area chart using the `px.area()` function. The arguments are:

- `x=month2019["Order Date"]`: This sets the x-axis values to the "Order Date" column from the month2019 DataFrame.
- `y=[month2019["Profit"], month2020["Profit"]]`: This sets the y-axis values to a list containing the "Profit" columns from both the month2019 and month2020 DataFrames. This will create two separate lines or areas in the chart, one for each year.

### Updating the Layout:

```
fig.update_layout(title={"text": "Profit by Month and Year", "x": 0.5, "xanchor": "center"})
```

```
fig.update_layout(legend={"title": "Years"})
```

These lines customize the layout of the chart:

- `title={"text": "Profit by Month and Year", "x": 0.5, "xanchor": "center"}`: This sets the title of the chart to "Profit by Month and Year" and centers it horizontally.
- `legend={"title": "Years"}`: This sets the title of the legend to "Years".

### Renaming Traces:

```
new_names = {"wide_variable_0": "2019", "wide_variable_1": "2020"}  
fig.for_each_trace(lambda t: t.update(name=new_names[t.name],  
                                     legendgroup=new_names[t.name],  
                                     hovertemplate=t.hovertemplate.replace(t.name,  
                                     new_names[t.name])))
```

**Note:** A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

This code renames the traces (lines or areas) in the chart to "2019" and "2020" for better readability. It also updates the legend and hover template to reflect the new names.

### Showing the Plot:

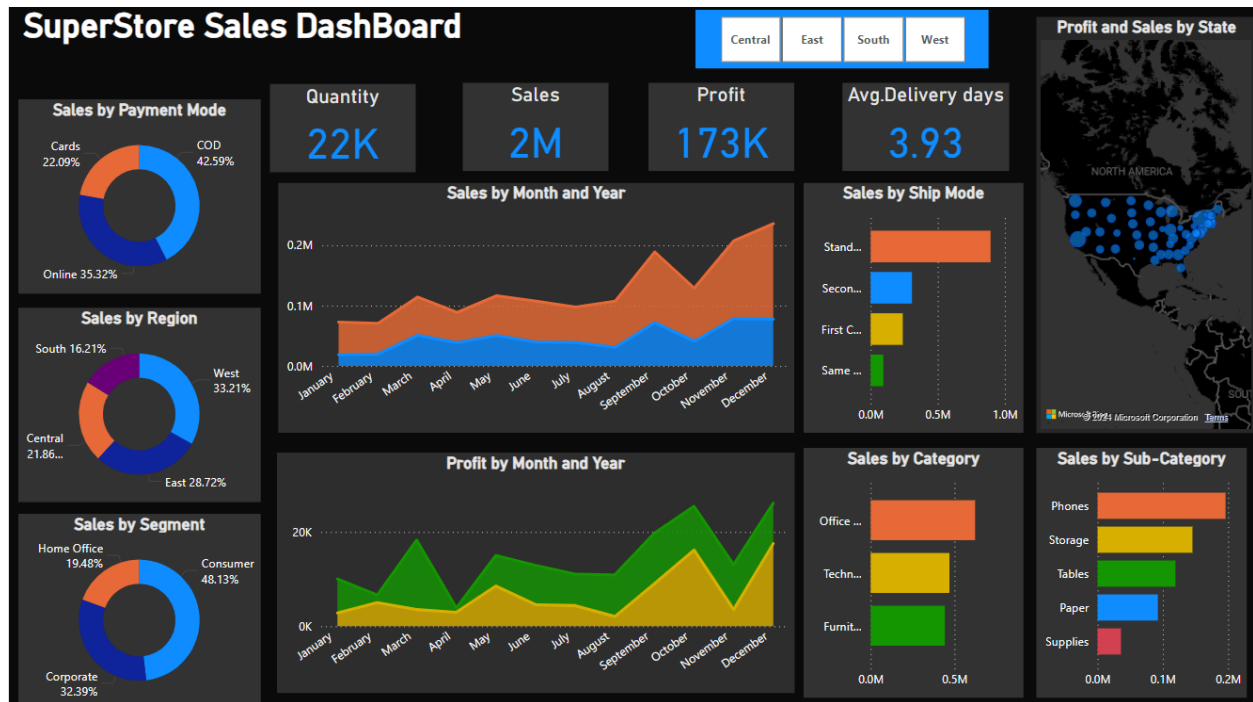
```
fig.show()
```

This line displays the created chart.

Now, we have to make a dashboard of the visualized data so we have to take the help of Power BI.

## 4. Presentation of the visualized dataset( Power BI)

The SuperStore Sales and Profit Dashboard comprehensively analyzes sales, profit, and performance metrics across different regions, segments, and categories. It allows users to gain insights into the business operations and understand key performance indicators like sales, profit, delivery times, and payment methods. This dashboard is interactive and segmented by different variables for easy data exploration.



### Key Metrics

- Quantity:**  
Displays the total quantity of products sold, which is **22K**.
- Sales:**  
Represents the total sales amount, showing a value of **2M**.
- Profit:**  
The total profit generated is **173K**.
- Average Delivery Days:**  
The average number of days for delivery is **3.93** days.



## Visualizations and Filters

### Sales by Payment Mode

This donut chart breaks down sales by different payment methods:

- **COD:** 42.59%
- **Online:** 35.32%
- **Cards:** 22.09%

### Sales by Region

This donut chart shows the sales distribution by region:

- **West:** 33.21%
- **East:** 28.72%
- **Central:** 21.86%
- **South:** 16.21%

### Sales by Segment

Sales are split among various customer segments:

- **Consumer:** 48.13%
- **Corporate:** 32.39%
- **Home Office:** 19.48%

### Sales by Month and Year

The area chart displays monthly sales trends over the year, with notable peaks in:

- **July** and **December**.

### Profit by Month and Year

This area chart highlights monthly profit trends, with a steady increase, especially during the latter half of the year, peaking in **December**.

### Sales by Ship Mode

This bar chart illustrates sales based on shipping methods:

- **Standard** is the most popular, followed by **Second Class** and **First Class**.

## **Sales by Category**

Top-performing categories include:

- **Office Supplies**
- **Technology**
- **Furniture**

## **Sales by Sub-Category**

A bar chart shows sales by sub-categories:

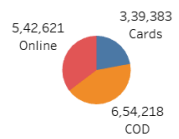
- **Phones** are the best-selling items, followed by **Storage** and **Tables**.

## **Sales by State**

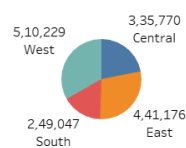
A geographical heat map shows the distribution of sales by state, indicating which regions contribute the most to the business.

## 5. Presentation of visualized Dataset by Tableau

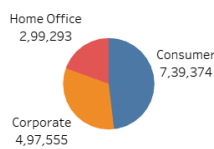
Sales by Payment Mode



Sales by Region



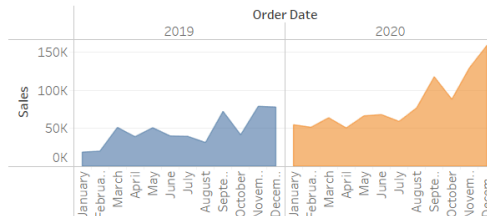
Sales by Segment



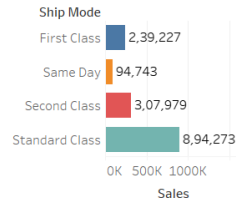
Super Store Sales and Profit Dashboard.

Profit	Quantity	Sales
172,921	21,562	1,536,222

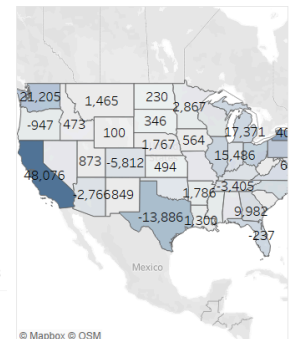
Sales by Month and Year



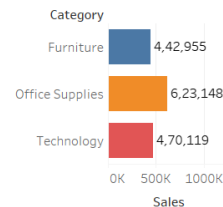
Sales by ShipMode



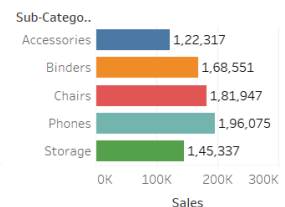
Sales and Profit by State



Sales by Category



Sales by Sub Category



The **Super Store Sales Dashboard** provides a comprehensive view of the store's sales, profit, and other key performance metrics. It displays data across various dimensions like payment modes, regions, segments, and categories, helping users track business performance and identify trends for decision-making.

### Key Metrics:

- **Total Sales:** \$1,536,222
- **Total Profit:** \$172,921
- **Total Quantity Sold:** 21,562 units
- **Components of the Dashboard**

#### 1. Sales by Payment Mode

A pie chart showing total sales across different payment methods:

- **Online:** \$542,621
- **Cards:** \$339,383
- **Cash on Delivery (COD):** \$654,218

This helps identify the most popular payment methods among customers.

## 2. Sales by Region

A pie chart comparing sales across different regions:

- **West:** \$5,10,229
- **Central:** \$3,35,770
- **South:** \$2,49,047
- **East:** \$4,41,176

This metric helps track which regions are driving the most revenue and where sales efforts can be focused.

## 3. Sales by Month and Year (2019-2020)

A dual-line chart showing sales trends over time for 2019 and 2020:

- Sales in 2019 saw fluctuations, with peaks around November.
- 2020 sales show a growing trend, especially after June.

This chart helps in understanding seasonal trends and sales growth over time.

## 4. Profit by Month and Year (2019-2020)

A line chart tracking monthly profit for 2019 and 2020:

- 2019 shows a spike in November, while 2020 sees stable profits after July.

This chart is useful for identifying profitable months and understanding the relationship between sales and profit over time.

## 5. Sales by Ship Mode

A bar chart comparing sales based on shipping methods:

- **First Class:** \$239,227
- **Same Day:** \$94,743
- **Second Class:** \$3,07,979
- **Standard Class:** \$8,94,273

This breakdown helps in analyzing shipping preferences and their impact on sales.

## 6. Sales and Profit by State

A geographical map showing the distribution of sales and profit by U.S. state:

- The size and color of the bubbles indicate the relative sales and profits across the states.

This visualization provides a spatial understanding of performance and can help in location-based decision-making.

## 7. Sales by Segment

A pie chart showing the sales for each customer segment:

- **Consumer:** \$739,374
- **Corporate:** \$497,555
- **Home Office:** \$299,293

This chart provides insights into which customer segments contribute the most to overall sales.

## 8. Sales by Category

A bar chart comparing sales by product category:

- **Furniture:** \$442,955
- **Office Supplies:** \$623,148
- **Technology:** \$470,119

This helps in understanding product categories' performance, aiding inventory and sales strategies.

## 9. Sales by Sub-Category

A detailed breakdown of sales by sub-categories like:

- **Accessories:** \$122,317
- **Binders:** \$168,551
- **Chairs:** \$181,947
- **Phones:** \$196,075
- **Storage:** \$145,337

## **Conclusion**

This is the report of 'super store sales and profit analysis' for two years 2019 and 2020 sales of different items like furniture, technology, and office supplies across all the states of the United States.

The documentation is made to analyze the growth of SuperStore Sales and Profit for the two years in which most of the customers used other modes as compared to card and online. The superstore has more sales in the west region and less sales in the south region as compared to the other two regions. They sold 22,000 units for \$ 2 million and earned a profit of \$1,75,000 in 2019, they bear the loss of \$9,411 and the sales have increased by \$87,779.

### **Recommendations for the increase in sales and customer satisfaction**

1. **Target Marketing:** Focus marketing efforts on the West region to further capitalize on its high sales potential.
2. **Customer Loyalty:** Implement strategies to retain and increase customer loyalty within the Consumer segment.
3. **Product Optimization:** Analyze the performance of different sub-categories to identify opportunities for product optimization or expansion.
4. **Shipping Efficiency:** Explore ways to improve shipping efficiency, particularly for Standard Class, to enhance customer satisfaction and reduce costs.
5. **Seasonal Trends:** Leverage seasonal trends to optimize inventory levels and marketing campaigns.

### **Additional Considerations:**

- **Customer Segmentation:** Consider segmenting customers based on demographics, purchase behavior, or other relevant factors to tailor marketing and sales strategies.
- **Competitive Analysis:** Conduct a thorough analysis of competitors to identify market trends, pricing strategies, and product offerings.
- **Data Quality:** Ensure the accuracy and completeness of the data used in the dashboard to make informed decisions.

By addressing these recommendations and considering the additional factors, the Super Store can further optimize its operations, improve profitability, and enhance customer satisfaction.

---

