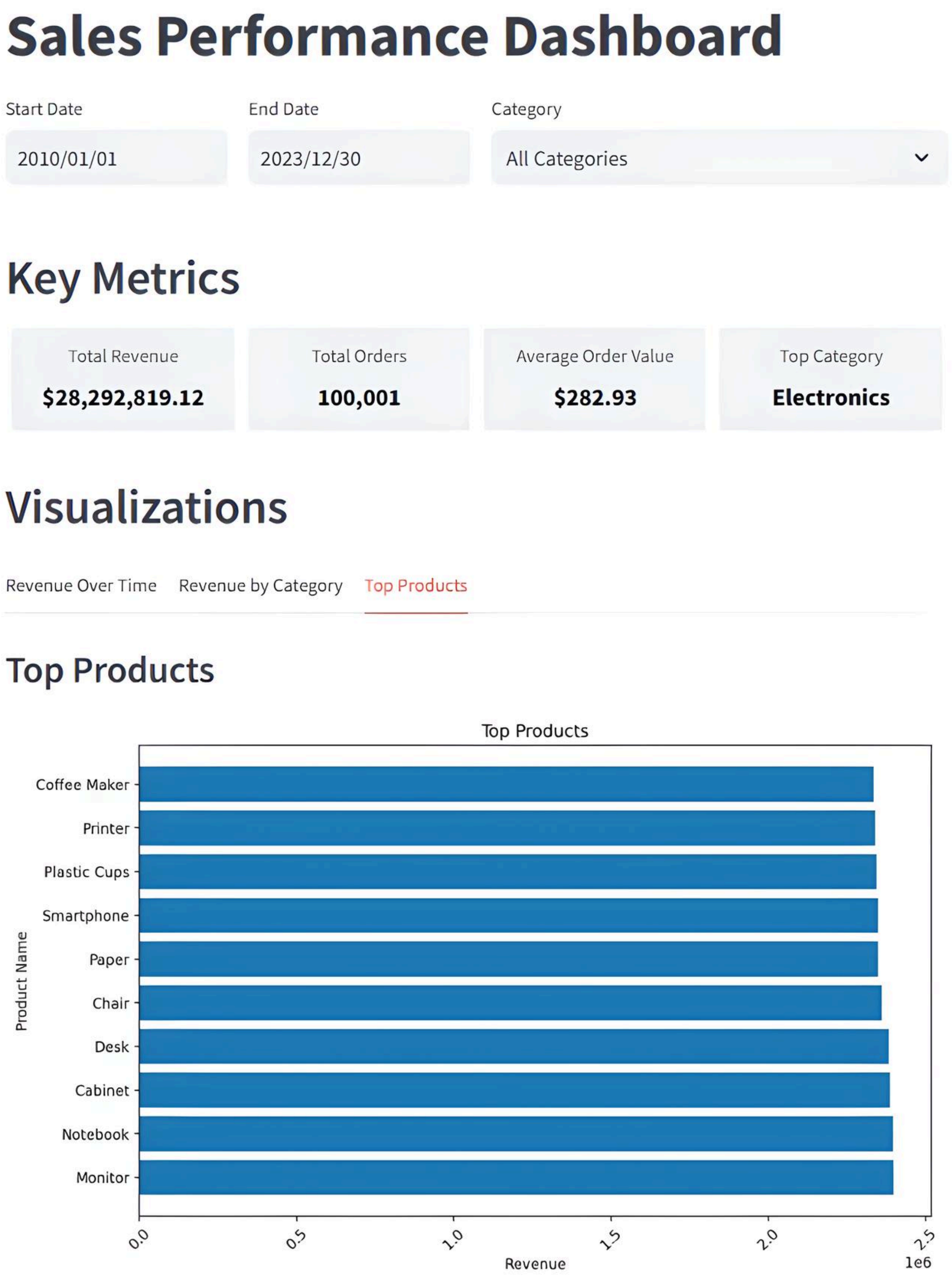
Building a Data Dashboard

Using the streamlit Python library

Thomas Reid

Jan 20, 2025 15 min read



Sales Performance Dashboard

Start Date: 2010/01/01

End Date: 2023/12/30

Category: All Categories

Key Metrics

Total Revenue	Total Orders	Average Order Value	Top Category
$28,292,819.12	100,001	$282.93	Electronics

Visualizations

Top Products

Image by Author

With source data from a Postgres database

As a Python data engineer for many years, one area I was not very involved in was the production of data dashboards. That all changed when Python-based libraries such as Streamlit, Gradio and Taipy came along.

With their introduction, python programmers had no excuses ***not*** to use them to craft nice-looking front-ends and dashboards.

Until then, the only other options were to use specialised tools like Tableau or AWS's Quicksight or—horror of horrors—get your hands dirty with CSS, HTML, and Javascript.

So, if you've never used one of these new Python-based graphical front-end libraries before, this article is for you as I'll be taking you through how to code up a data dashboard using one of the most popular libraries for this purpose called `Streamlit.`

> My intention is that this will be the first part of a series of articles on developing a data dashboard using three of the most popular Python-based GUI libraries. In addition to this one, I also plan to release articles on Gradio and Taipy, so look out for those. As much as possible I'll try to replicate the same layout and functionality in each dashboard. I'll use the exact same data for all three too, albeit in different formats e.g. a CSV, database etc …
>
> Please also note that I have no connection or affiliation with Streamlit/Snowflake, Postgres or any other company or tool mentioned in this post.

What is Streamlit?

Founded in 2018 by Adrien Treuille, Amanda Kelly, and Thiago Teixeira, Streamlit quickly gained popularity among data scientists and machine learning engineers when it introduced its open-source Python framework to simplify the creation of interactive data applications.

In March 2022, Snowflake, a Data Cloud company, acquired Streamlit and its capabilities were integrated into the Snowflake ecosystem to enhance data application development.

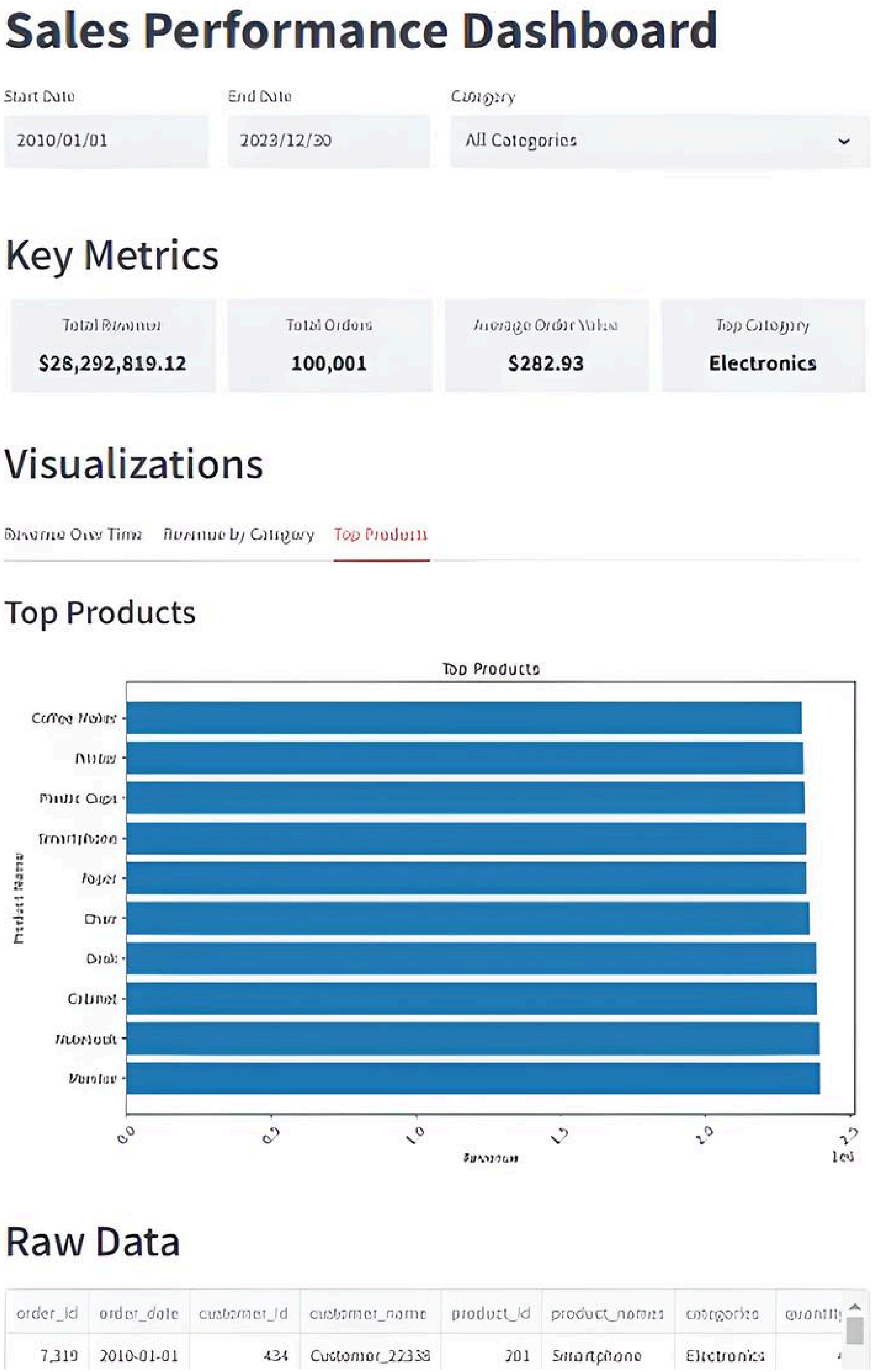
Streamlit's open-source framework has been widely adopted, with over 8 million downloads and more than 1.5 million applications built using the platform. An active community of developers and contributors continues to play a significant role in its ongoing development and success.

What we'll develop

We're going to develop a data dashboard. Our source data for the dashboard will be in a single Postgres database table and contain 100,000 synthetic sales records.

To be honest, the actual source of the data isn't ***that*** important. It could just as easily be a text or CSV file, SQLite, or any database you can connect to. I chose Postgres because I have a copy on my local PC, and it's convenient for me to use.

This is what our final dashboard will look like.



Sales Performance Dashboard

Start Date	End Date	Category
2010/01/01	2023/12/30	All Categories

Key Metrics

Total Revenue	Total Orders	Average Order Value	Top Category
$28,292,819.12	100,001	$282.93	Electronics

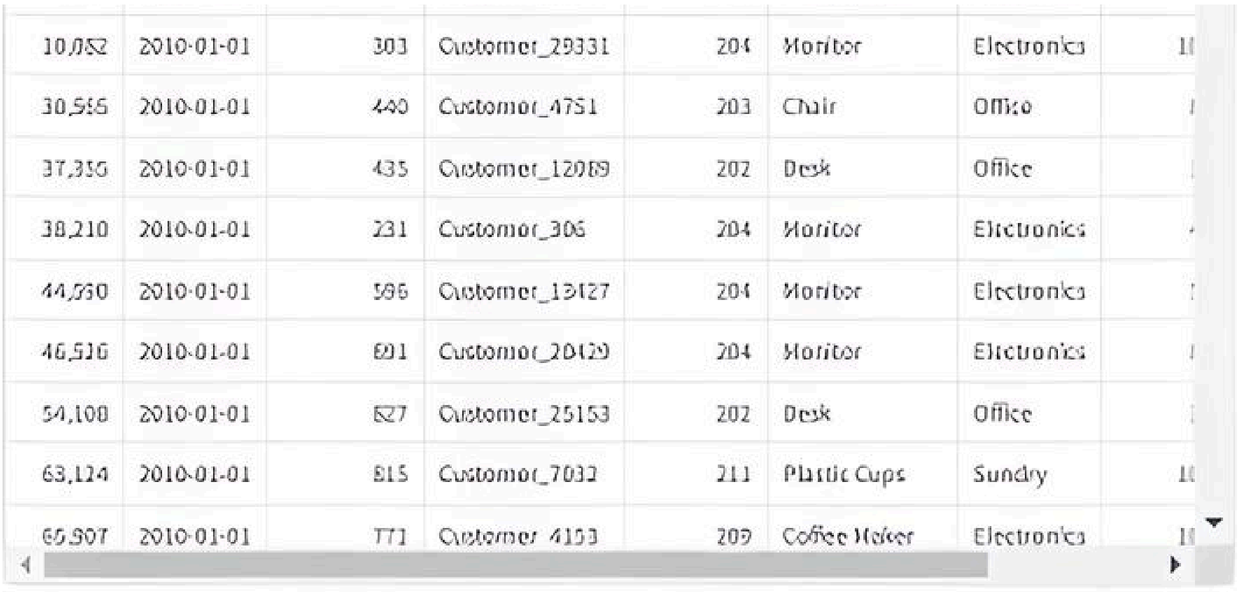
Visualizations

Revenue Over Time Revenue by Category Top Products

Top Products

Raw Data

order_id	order_date	customer_id	customer_name	product_id	product_names	categories	quantity
7,319	2010-01-01	434	Customer_22338	201	Smartphone	Electronics	[illegible]



10,052	2010-01-01	303	Customer_29331	204	Monitor	Electronics	[illegible]
30,566	2010-01-01	440	Customer_4751	203	Chair	Office	[illegible]
37,356	2010-01-01	435	Customer_12089	202	Desk	Office	[illegible]
38,210	2010-01-01	231	Customer_306	204	Monitor	Electronics	[illegible]
44,930	2010-01-01	596	Customer_13427	204	Monitor	Electronics	[illegible]
46,516	2010-01-01	691	Customer_20429	204	Monitor	Electronics	[illegible]
54,100	2010-01-01	627	Customer_25153	202	Desk	Office	[illegible]
63,124	2010-01-01	815	Customer_7032	211	Plastic Cups	Sundry	[illegible]
65,907	2010-01-01	771	Customer_4153	209	Coffee Maker	Electronics	[illegible]

Image by Author

There are four main sections.

- The top row allows the user to choose specific start and end dates and/or product categories via date pickers and a drop-down list, respectively.
- The second row – Key metrics – shows a top-level summary of the chosen data.
- The Visualisation section allows the user to select one of three graphs to display the input data set.
- The raw data section is exactly what it says. This is a tabular representation of the chosen data, effectively viewing the underlying Postgres database table data.

Using the dashboard is easy. Initially, stats for the whole data set are displayed. The user can then narrow the data focus using the 3 choice fields at the top of the display. The graphs, key metrics and raw data sections dynamically change to reflect what the user has chosen.

The underlying data

As mentioned, the dashboard's source data is contained in a single Postgres database table. The data is a set of 100,000 synthetic sales-related data records. Here is the Postgres table creation script for reference.

```
CREATE TABLE IF NOT EXISTS public.sales_data
(
    order_id integer NOT NULL,
    order_date date,
    customer_id integer,
    customer_name character varying(255) COLLATE pg_catalog."default",
    product_id integer,
    product_names character varying(255) COLLATE pg_catalog."default",
    categories character varying(100) COLLATE pg_catalog."default",
    quantity integer,
    price numeric(10,2),
    total numeric(10,2)
)
```

And here is some Python code you can use to generate a data set for yourself. Make sure both numpy and polars libraries are installed first

```
# generate the 1m record CSV file
#
import polars as pl
import numpy as np
from datetime import datetime, timedelta

def generate(nrows: int, filename: str):
    names = np.asarray(
        [
            "Laptop",
            "Smartphone",
            "Desk",
            "Chair",
            "Monitor",
            "Printer",
            "Paper",
            "Pen",
            "Notebook",
            "Coffee Maker",
            "Cabinet",
            "Plastic Cups",
        ]
    )

    categories = np.asarray(
        [
            "Electronics",
            "Electronics",
            "Office",
            "Office",
            "Electronics",
            "Electronics",
            "Stationery",
            "Stationery",
            "Stationery",
            "Electronics",
            "Office",
            "Sundry",
        ]
    )

    product_id = np.random.randint(len(names), size=nrows)
    quantity = np.random.randint(1, 11, size=nrows)
    price = np.random.randint(199, 10000, size=nrows) / 100

    # Generate random dates between 2010-01-01 and 2023-12-31
    start_date = datetime(2010, 1, 1)
    end_date = datetime(2023, 12, 31)
    date_range = (end_date - start_date).days

    # Create random dates as np.array and convert to string format
    order_dates = np.array([(start_date + timedelta(days=np.random.randint(

    # Define columns
    columns = {
        "order_id": np.arange(nrows),
        "order_date": order_dates,
        "customer_id": np.random.randint(100, 1000, size=nrows),
        "customer_name": [f"Customer_{i}" for i in np.random.randint(2**15,
        "product_id": product_id + 200,
        "product_names": names[product_id],
        "categories": categories[product_id],
        "quantity": quantity,
        "price": price,
        "total": price * quantity,
    }

    # Create Polars DataFrame and write to CSV with explicit delimiter
    df = pl.DataFrame(columns)
    df.write_csv(filename, separator=',',include_header=True)  # Ensure comm

# Generate 100,000 rows of data with random order_date and save to CSV
generate(100_000, "/mnt/d/sales_data/sales_data.csv")
```

Setting up our development environment

Before we get to the example code, let's set up a separate development environment. That way, what we do won't interfere with other versions of libraries, programming, etc… we might have on the go for other projects we're working on.

I use Miniconda for this, but you can use whatever method suits you best.

If you want to go down the Miniconda route and don't already have it, you must install Miniconda first. Get it using this link,

Miniconda – Anaconda documentation

Once the environment is created, switch to it using the `activate` command, and then `pip install` our required Python libraries.

```
#create our test environment
(base) C:Usersthoma>conda create -n streamlit_test python=3.12 -y


# Now activate it
(base) C:Usersthoma>conda activate streamlit_test


# Install python libraries, etc ...
(streamlit_test) C:Usersthoma>pip install streamlit pandas matplotlib psycop
```

The Code

I'll split the code up into sections and explain each one along the way.

```
#
# Streamlit equivalent of final Gradio app
#
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import psycopg2
from psycopg2 import sql
from psycopg2 import pool

# Initialize connection pool
try:
    connection_pool = psycopg2.pool.ThreadedConnectionPool(
        minconn=5,
        maxconn=20,
        dbname="postgres",
        user="postgres",
        password="postgres",
        host="localhost",
        port="5432"
    )
except psycopg2.Error as e:
        st.error(f"Error creating connection pool: {e}")

def get_connection():
    try:
        return connection_pool.getconn()
    except psycopg2.Error as e:
        st.error(f"Error getting connection from pool: {e}")
        return None

def release_connection(conn):
    try:
        connection_pool.putconn(conn)
    except psycopg2.Error as e:
        st.error(f"Error releasing connection back to pool: {e}")
```

We start by importing all the external libraries we'll need. Next, we set up a **ThreadedConnectionPool** that allows **** multiple threads to share a pool of database connections. Two helper functions follow, one to get a database connection and the other to release it. This is overkill for a simple single-user app but essential for handling multiple simultaneous users or threads accessing the database in a web app environment.

```
def get_date_range():
    conn = get_connection()
    if conn is None:
        return None, None
    try:
        with conn.cursor() as cur:
            query = sql.SQL("SELECT MIN(order_date), MAX(order_date) FROM pu
            cur.execute(query)
            return cur.fetchone()
    finally:
        release_connection(conn)

def get_unique_categories():
    conn = get_connection()
```

```
        if conn is None:
            return []
        try:
            with conn.cursor() as cur:
                query = sql.SQL("SELECT DISTINCT categories FROM public.sales_da
                cur.execute(query)
                return [row[0].capitalize() for row in cur.fetchall()]
        finally:
            release_connection(conn)

    def get_dashboard_stats(start_date, end_date, category):
        conn = get_connection()
        if conn is None:
            return None
        try:
            with conn.cursor() as cur:
                query = sql.SQL("""
                    WITH category_totals AS (
                        SELECT
                            categories,
                            SUM(price * quantity) as category_revenue
                        FROM public.sales_data
                        WHERE order_date BETWEEN %s AND %s
                        AND (%s = 'All Categories' OR categories = %s)
                        GROUP BY categories
                    ),
                    top_category AS (
                        SELECT categories
                        FROM category_totals
                        ORDER BY category_revenue DESC
                        LIMIT 1
                    ),
                    overall_stats AS (
                        SELECT
                            SUM(price * quantity) as total_revenue,
                            COUNT(DISTINCT order_id) as total_orders,
                            SUM(price * quantity) / COUNT(DISTINCT order_id) as
                        FROM public.sales_data
                        WHERE order_date BETWEEN %s AND %s
                        AND (%s = 'All Categories' OR categories = %s)
                    )
                    SELECT
                        total_revenue,
                        total_orders,
                        avg_order_value,
                        (SELECT categories FROM top_category) as top_category
                    FROM overall_stats
                """)
                cur.execute(query, [start_date, end_date, category, category,
                                    start_date, end_date, category, category])
                return cur.fetchone()
        finally:
            release_connection(conn)
```

The **get_date_range** function executes the SQL query to find the range of dates (`MIN` and `MAX`) in the `order_date` column and returns the two dates as a tuple: `(start_date, end_date)` .

The **get_unique_categories** function runs an SQL query to fetch unique values from the `categories` column. It capitalizes the category names (first letter uppercase) before returning them as a list.

The **get_dashboard_stats** function executes a SQL query with the following parts:

- **`category_totals`** : Calculates total revenue for each category in the given date range.
- **`top_category`** : Finds the category with the highest revenue.
- **`overall_stats`** : Computes overall statistics:
- Total revenue (`SUM(price * quantity)`).
- Total number of unique orders (`COUNT(DISTINCT order_id)`).
- Average order value (total revenue divided by total orders).

It returns a single row containing:

- `total_revenue` : Total revenue in the specified period.
- `total_orders` : Number of distinct orders.
- `avg_order_value` : Average revenue per order.
- `top_category` : The category with the highest revenue.

```
def get_plot_data(start_date, end_date, category):
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT DATE(order_date) as date,
                       SUM(price * quantity) as revenue
                FROM public.sales_data
                WHERE order_date BETWEEN %s AND %s
                  AND (%s = 'All Categories' OR categories = %s)
                GROUP BY DATE(order_date)
                ORDER BY date
            """)
            cur.execute(query, [start_date, end_date, category, category])
            return pd.DataFrame(cur.fetchall(), columns=['date', 'revenue'])
    finally:
        release_connection(conn)

def get_revenue_by_category(start_date, end_date, category):
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT categories,
                       SUM(price * quantity) as revenue
                FROM public.sales_data
                WHERE order_date BETWEEN %s AND %s
                  AND (%s = 'All Categories' OR categories = %s)
                GROUP BY categories
                ORDER BY revenue DESC
            """)
            cur.execute(query, [start_date, end_date, category, category])
            return pd.DataFrame(cur.fetchall(), columns=['categories', 'reve
    finally:
        release_connection(conn)

def get_top_products(start_date, end_date, category):
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT product_names,
                       SUM(price * quantity) as revenue
                FROM public.sales_data
                WHERE order_date BETWEEN %s AND %s
                  AND (%s = 'All Categories' OR categories = %s)
                GROUP BY product_names
                ORDER BY revenue DESC
                LIMIT 10
            """)
            cur.execute(query, [start_date, end_date, category, category])
            return pd.DataFrame(cur.fetchall(), columns=['product_names', 'r
    finally:
        release_connection(conn)

def get_raw_data(start_date, end_date, category):
    conn = get_connection()
    if conn is None:
        return pd.DataFrame()
    try:
        with conn.cursor() as cur:
            query = sql.SQL("""
                SELECT
                    order_id, order_date, customer_id, customer_name,
                    product_id, product_names, categories, quantity, price,
                    (price * quantity) as revenue
                FROM public.sales_data
                WHERE order_date BETWEEN %s AND %s
                  AND (%s = 'All Categories' OR categories = %s)
                ORDER BY order_date, order_id
            """)
            cur.execute(query, [start_date, end_date, category, category])
            return pd.DataFrame(cur.fetchall(), columns=[desc[0] for desc ir
    finally:
        release_connection(conn)

def plot_data(data, x_col, y_col, title, xlabel, ylabel, orientation='v'):
    fig, ax = plt.subplots(figsize=(10, 6))
    if not data.empty:
        if orientation == 'v':
            ax.bar(data[x_col], data[y_col])
        else:
            ax.barh(data[x_col], data[y_col])
        ax.set_title(title)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        plt.xticks(rotation=45)
    else:
        ax.text(0.5, 0.5, "No data available", ha='center', va='center')
    return fig
```

The **get_plot_data** function fetches daily revenue within the given date range and category. It retrieves data grouped by the day (`DATE(order_date)`) and calculates daily revenue (`SUM(price * quantity)`), then returns a Pandas DataFrame with columns: `date` (the day) and `revenue` (total revenue for that day).

The **get_revenue_by_category** function fetches revenue totals grouped by category within the specified date range. It groups data by `categories` and calculates revenue for each category (`SUM(price * quantity)`), orders the results by revenue in descending order and returns a Pandas DataFrame with columns: `categories` (category name) and `revenue` (total revenue for the category).

The **get_top_products** function retrieves the top 10 products by revenue within the given date range and category. It groups data by `product_names` and calculates revenue for each product (`SUM(price * quantity)`), orders the products by revenue in descending order and limits results to the top 10 before returning a Pandas DataFrame with columns: `product_names` (product name) and `revenue` (total revenue for the product).

The **get_raw_data** function fetches raw transaction data within the specified date range and category.

The **plot_data** function takes in some data (in a pandas DataFrame) and the names of the columns you want to plot on the x- and y-axes. It then creates a bar chart – either vertical or horizontal, depending on the chosen orientation – labels the axes, adds a title, and returns the finished chart (a Matplotlib Figure). If the data is empty, it just displays a "No data available" message instead of trying to plot anything.

```
# Streamlit App
st.title("Sales Performance Dashboard")

# Filters
with st.container():
    col1, col2, col3 = st.columns([1, 1, 2])
    min_date, max_date = get_date_range()
    start_date = col1.date_input("Start Date", min_date)
    end_date = col2.date_input("End Date", max_date)
    categories = get_unique_categories()
    category = col3.selectbox("Category", ["All Categories"] + categories)
```

```
# Custom CSS for metrics
st.markdown("""
    <style>
    .metric-row {
        display: flex;
        justify-content: space-between;
        margin-bottom: 20px;
    }
    .metric-container {
        flex: 1;
        padding: 10px;
        text-align: center;
        background-color: #f0f2f6;
        border-radius: 5px;
        margin: 0 5px;
    }
    .metric-label {
        font-size: 14px;
        color: #555;
        margin-bottom: 5px;
    }
    .metric-value {
        font-size: 18px;
        font-weight: bold;
        color: #0e1117;
    }
    </style>
""", unsafe_allow_html=True)

# Metrics
st.header("Key Metrics")
stats = get_dashboard_stats(start_date, end_date, category)
if stats:
    total_revenue, total_orders, avg_order_value, top_category = stats
else:
    total_revenue, total_orders, avg_order_value, top_category = 0, 0, 0, "N

# Custom metrics display
metrics_html = f"""
```

```
<div class="metric-row">
    <div class="metric-container">
        <div class="metric-label">Total Revenue</div>
        <div class="metric-value">${total_revenue:,.2f}</div>
    </div>
    <div class="metric-container">
        <div class="metric-label">Total Orders</div>
        <div class="metric-value">{total_orders:,}</div>
    </div>
    <div class="metric-container">
        <div class="metric-label">Average Order Value</div>
        <div class="metric-value">${avg_order_value:,.2f}</div>
    </div>
    <div class="metric-container">
        <div class="metric-label">Top Category</div>
        <div class="metric-value">{top_category}</div>
    </div>
</div>
"""
st.markdown(metrics_html, unsafe_allow_html=True)
```

This code section creates the main structure for displaying the key metrics in the Streamlit dashboard. It:

1. **Sets up the page title**: "Sales Performance Dashboard."
2. **Presents filters** for start/end dates and category selection.
3. **Retrieves metrics** (such as total revenue, total orders, etc.) for the chosen filters from the database.
4. **Applies custom CSS** to style these metrics in a row of boxes with labels and values.
5. **Displays the metrics** within an HTML block, ensuring each metric gets its own styled container.

```
# Visualization Tabs
st.header("Visualizations")
tabs = st.tabs(["Revenue Over Time", "Revenue by Category", "Top Products"])

# Revenue Over Time Tab
with tabs[0]:
    st.subheader("Revenue Over Time")
    revenue_data = get_plot_data(start_date, end_date, category)
    st.pyplot(plot_data(revenue_data, 'date', 'revenue', "Revenue Over Time'

# Revenue by Category Tab
with tabs[1]:
    st.subheader("Revenue by Category")
    category_data = get_revenue_by_category(start_date, end_date, category)
    st.pyplot(plot_data(category_data, 'categories', 'revenue', "Revenue by

# Top Products Tab
with tabs[2]:
    st.subheader("Top Products")
    top_products_data = get_top_products(start_date, end_date, category)
    st.pyplot(plot_data(top_products_data, 'product_names', 'revenue', "Top
```

This section adds a header titled "Visualizations" to this part of the dashboard. It creates three tabs, each of which displays a different graphical representation of the data:

Tab 1: Revenue Over Time

- Fetches **revenue data grouped by date** for the given filters using `get_plot_data()`.
- Calls `plot_data()` to generate a **bar chart** of revenue over time, with dates on the x-axis and revenue on the y-axis.
- Displays the chart in the first tab.

Tab 2: Revenue by Category

- Fetches **revenue grouped by category** using `get_revenue_by_category()` .
- Calls `plot_data()` to create a **bar chart** of revenue by category.
- Displays the chart in the second tab.

Tab 3: Top Products

- Fetches **top 10 products by revenue** for the given filters using `get_top_products()` .
- Calls `plot_data()` to create a **horizontal bar chart** (indicated by `orientation='h'`).
- Displays the chart in the third tab.

```
st.header("Raw Data")

raw_data = get_raw_data(
    start_date=start_date,
    end_date=end_date,
    category=category
)

# Remove the index by resetting it and dropping the old index
raw_data = raw_data.reset_index(drop=True)

st.dataframe(raw_data,hide_index=True)

# Add spacing
st.write("")
```

The final section displays the raw data in a dataframe. The user is able to scroll up and down as required to see all records available.

An empty `st.write("")` is added at the end to provide spacing for better visual alignment.

Running the App

Let’s say you save your code into a file called app.py. You can run it using this from the command line,

```
(streamlit_test) C:Usersthoma> python -m streamlit run app.py
```

If everything works as expected, you will see this after you run the above command.

```

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.0.59:8501
```

Click on the Local URLs shown, and a browser screen should appear with the Streamlit app running.

Summary

In this article, I’ve attempted to provide a comprehensive guide to building an interactive sales performance dashboard using Streamlit with a Postgres database table as its source data.

Streamlit is a modern, Python-based open-source framework that simplifies the creation of data-driven dashboards and applications. The dashboard I developed allows users to filter data by date ranges and product categories, view key metrics such as total revenue and top-performing categories, explore visualizations like revenue trends and top products, and navigate through raw data with pagination.

This guide includes a complete implementation, from setting up a Postgres database with sample data to creating Python functions for querying data, generating plots, and handling user input. This step-by-step approach demonstrates how to leverage Streamlit's capabilities to create user-friendly and dynamic dashboards, making it ideal for data engineers and scientists who want to build interactive data applications.

Although I used Postgres for my data, it should be straightforward to modify the code to use a CSV file or any other relational database management system (RDBMS), such as SQLite, as your data source.

> _That's all from me for now. I hope you found this article useful. If you did, please check out my profile page at this link. From there, you can see my other published stories and subscribe to get notified when I post new content._

If you liked this content, Medium thinks you'll find these articles interesting, too.

> **Speed up Pandas code with Numpy**
>
> **Introducing Deepseek Artifacts**