IBM GRM PROJECT REPORT

ON

Data Integration and Transformation Using Box API and IBM DB2

Submitted by:

Annem Siva Pranav Reddy,
Aren Dias,
Garret Fernandez,
Kasibhatta Suprabhat

Objective

The goal of this project is to process and analyze e-commerce transactional data using Apache Spark, generate a consolidated order summary and product sales insights, and store the results in an IBM Db2 database for further use, such as business intelligence dashboards or reporting.

Tools & Technologies:

- Apache Spark (PySpark) – for distributed data processing
- IBM Db2 on Cloud – as the target relational database
- CSV Files – cleaned source data for orders, products, users, etc.
- Spark JDBC – to write results to Db2
- Google Colab / Jupyter Notebook – used for development and testing

Source Datasets

All source data was pre-cleaned and stored as CSV files in `/content/Source_Data/`.

Dataset Name	Description
cleaned_orders.csv	Basic order details
cleaned_order_items.csv	Items associated with each order
cleaned_users.csv	Customer/user information
cleaned_products.csv	Product catalog
cleaned_distribution_centers.csv	Details of product distribution centers
cleaned_inventory_items.csv	Mapping of inventory items to products

About our Dataset:

The dataset used in this project comes from a retail business and contains information about users, products, orders, inventory, and distribution centers. It includes the following CSV files:

- **users.csv** – Details about customers, such as their name, age, gender, location, and how they found the website.
- **orders.csv** – Basic information about each order, including when it was placed, shipped, delivered, or returned.
- **order_items.csv** – Item-level details for each order, showing which products were bought and their status.
- **products.csv** – Information about products like name, category, brand, price, and the distribution center they belong to.
- **inventory_items.csv** – Tracks each product in inventory, including when it was added and sold.
- **distribution_centers.csv** – Locations of the warehouses where products are stored, with coordinates.
- **events.csv** – Logs of user activity on the website, including browser used, city, and type of action taken.

These files together help us analyze how customers interact with the system, what they buy, and how products move through the supply chain.

Our Work

Step 1: Connecting to Box and Handling Files

This Python script demonstrates a complete solution for interacting with Box, a cloud content management platform, using OAuth2 refresh tokens. The script supports both downloading and uploading files between local storage and Box folders using the official boxsdk.

- **Environment Setup and Configuration:** The script starts by loading environment variables using the dotenv package. These variables store sensitive credentials and configuration details.

```python
import io
import os
from dotenv import load_dotenv
from boxsdk import OAuth2, Client
from boxsdk.exception import BoxAPIException

load_dotenv()

BOX_CLIENT_ID = os.getenv('BOX_CLIENT_ID')
BOX_CLIENT_SECRET = os.getenv('BOX_CLIENT_SECRET')
BOX_REFRESH_TOKEN = os.getenv('BOX_REFRESH_TOKEN')

BOX_DOWNLOAD_FOLDER_ID = os.getenv('BOX_DOWNLOAD_FOLDER_ID')
BOX_UPLOAD_FOLDER_ID = os.getenv('BOX_UPLOAD_FOLDER_ID')

SOURCE_DATA_DIR = os.getenv('SOURCE_DATA_DIR')
```

- **Authenticating with Box Using OAuth2:** Box uses OAuth2 for secure access. This function handles the authentication using a refresh token, allowing for token renewal without user interaction.

```python
def get_box_client():
    if not all([BOX_CLIENT_ID, BOX_CLIENT_SECRET, BOX_REFRESH_TOKEN]):
        print("ERROR: One or more required Box OAuth2 environment variables (CLIENT_ID, CLIENT_SECRET, REFRESH_TOKEN) are missing from .env.")
        print("Ensure BOX_CLIENT_ID, BOX_CLIENT_SECRET are set from your Box App config, and BOX_REFRESH_TOKEN is obtained via get_box_tokens.py.")
        return None

    try:
        oauth = OAuth2(
            client_id=BOX_CLIENT_ID,
            client_secret=BOX_CLIENT_SECRET,
            access_token=None,
            refresh_token=BOX_REFRESH_TOKEN,
            store_tokens=store_tokens,
        )

        return Client(oauth)
    except Exception as e:
        print(f"Error authenticating with Box using OAuth2 Refresh Token: {e}")
        print("Possible causes: Invalid BOX_CLIENT_ID, BOX_CLIENT_SECRET, or expired/invalid BOX_REFRESH_TOKEN.")
        print("If BOX_REFRESH_TOKEN is invalid, re-run get_box_tokens.py to get a new one.")
        return None
```

- **Downloading Files from a Box Folder:** The download_box_files() function connects to a specified Box folder, filters files by prefix, and downloads them locally.

```python
def download_box_files(folder_id, target_dir, file_prefix_filter=None):

    client = get_box_client()
    if client is None:
        return []

    try:
        folder = client.folder(folder_id).get()
        print(f"Accessing Box folder '{folder.name}' (ID: {folder_id})...")
    except BoxAPIException as e:
        print(f"ERROR: Box API access failed for folder ID {folder_id}: {e.status} - {e.message}")
        if e.status == 404:
            print("Please ensure the folder ID is correct and the Box application has access.")
        elif e.status == 401:
            print("Authentication error. Please check your Box OAuth2 credentials and application authorization.")
        return []
    except Exception as e:
        print(f"An unexpected error occurred while accessing Box folder ID {folder_id}: {e}")
        return []

    os.makedirs(target_dir, exist_ok=True)
    downloaded_files = []
```

- **Uploading Files to a Box Folder:** The upload_file_to_box() function allows uploading a file from the local system to a specified Box folder.

```python
def upload_file_to_box(folder_id, file_path):

    client = get_box_client()
    if client is None:
        return None

    if not os.path.exists(file_path):
        print(f"ERROR: Local file not found for upload: {file_path}")
        return None

    file_name = os.path.basename(file_path)
    try:
        folder = client.folder(folder_id).get()
        uploaded_file = folder.upload(file_path, file_name=file_name)
        print(f"Uploaded file '{file_name}' to Box folder '{folder.name}' (ID: {folder_id}) with file ID: {uploaded_file.id}")
        return uploaded_file
    except BoxAPIException as e:
        print(f"ERROR: Box API upload failed for file '{file_name}' to folder ID {folder_id}: {e.status} - {e.message}")
        if e.status == 404:
            print("Please ensure the upload folder ID is correct and exists.")
        elif e.status == 403:
            print("Permission denied. Please check your Box application's permissions for write access.")
        return None
    except Exception as e:
        print(f"An unexpected error occurred during upload of '{file_name}': {e}")
        return None
```

- **Script Execution and Workflow:** When run directly, the script creates the local data directory and downloads only files with a specific prefix (cleaned_) from Box.

```python
if __name__ == "__main__":

    os.makedirs(SOURCE_DATA_DIR, exist_ok=True)

    print("\n--- Attempting to download cleaned CSV files from Box using OAuth2 Refresh Token ---")
    download_box_files(
        folder_id=BOX_DOWNLOAD_FOLDER_ID,
        target_dir=SOURCE_DATA_DIR,
        file_prefix_filter='cleaned_'
    )
```

Step 2: Connecting to IBM Db2 and Creating a Table

This Python script establishes a secure connection to an IBM Db2 cloud database and performs the following:

1. Creates a table (if not already present),
2. Inserts data into it from a local CSV file.

- **Environment Configuration:** The script reads database credentials and connection details from environment variables, with hardcoded defaults as fallbacks.

```
import os

DB2_HOSTNAME = os.getenv('DB2_HOSTNAME', '...cloud')
DB2_UID = os.getenv('DB2_UID', '...')
DB2_PWD = os.getenv('DB2_PWD', '...')
DB2_DATABASE = os.getenv('DB2_DATABASE', 'bludb')
DB2_PORT = os.getenv('DB2_PORT', '30756')
```

- **Establishing Connection to Db2:** The get_db2_connection() function builds a Data Source Name (DSN) string and attempts a secure connection over SSL.

```python
def get_db2_connection():
    dsn = (
        f"DATABASE={DB2_DATABASE};"
        f"HOSTNAME={DB2_HOSTNAME};"
        f"PORT={DB2_PORT};"
        f"PROTOCOL=TCPIP;"
        f"UID={DB2_UID};"
        f"PWD={DB2_PWD};"
        "SECURITY=SSL;"
    )
    try:
        conn = ibm_db.connect(dsn, "", "")
        print("Successfully connected to Db2 database")
        return conn
    except Exception as e:
        print("Connection failed:", e)
        return None
```

- **Creating and Populating the Table:** The create_and_insert_distribution_centers() function handles two main tasks:
  1. Creating the distribution_centers table.
  2. Inserting records from a CSV file.

```python
def create_and_insert_distribution_centers(conn, csv_path):
    create_table_sql = """
    CREATE TABLE distribution_centers (
        id INTEGER,
        name VARCHAR(100),
        latitude DOUBLE,
        longitude DOUBLE
    )
    """
    try:
        ibm_db.exec_immediate(conn, create_table_sql)
        print("Table 'distribution_centers' created (if it didn't exist).")
    except Exception as e:
        print(f"Table might already exist or error occurred: {e}")
```

Next, the CSV file is read and inserted row by row:

```
try:
    df = pd.read_csv(csv_path)
    print(f"Inserting {len(df)} rows into distribution_centers...")
    for i, row in df.iterrows():
        insert_sql = f"""
        INSERT INTO distribution_centers (id, name, latitude, longitude)
        VALUES ({int(row['id'])}, '{row['name'].replace("'", "''")}', {row['latitude']}, {ro
        """
        ibm_db.exec_immediate(conn, insert_sql)
    print("Data inserted successfully into 'distribution_centers'.")
except Exception as e:
    print(f"Error inserting data: {e}")
```

Step 3: Uploading Cleaned CSVs to IBM Db2 using PySparkLoad CSV Data:
This script uses Apache Spark (PySpark) to:

1. Load CSV files,
2. Preview and transform them using Spark DataFrames,
3. Upload them efficiently to IBM Db2 using JDBC.

It is designed for batch-oriented data ingestion into a cloud database.

- **Environment Configuration & Dependencies:** The script uses environment variables for all credentials and file paths. It loads these using dotenv and also initializes Spark using findspark.

```
import os
import findspark
from pyspark.sql import SparkSession
from dotenv import load_dotenv

load_dotenv()
findspark.init()
```

- **Initializing the Spark Session:** The function get_spark_session() sets up a SparkSession and configures it with the IBM Db2 JDBC driver.

```
def get_spark_session():
    abs_jdbc_path = os.path.abspath(JDBC_DRIVER_PATH)
    if not os.path.exists(abs_jdbc_path):
        print("ERROR: JDBC driver not found at", abs_jdbc_path)
        exit(1)

    spark = SparkSession.builder \
        .appName("Upload Cleaned CSVs to DB2") \
        .config("spark.jars", abs_jdbc_path) \
        .config("spark.driver.extraClassPath", abs_jdbc_path) \
        .getOrCreate()
    return spark
```

- **Configuring JDBC Properties:** Connection properties for writing to Db2 are returned by get_jdbc_properties()

```
def get_jdbc_properties():
    return {
        "user": DB2_UID,
        "password": DB2_PWD,
        "driver": "com.ibm.db2.jcc.DB2Driver",
        "sslConnection": "true",
        "currentSchema": DB2_SCHEMA,
        "batchsize": "200"
    }
```

The script uses secure SSL connection and batch insert to improve performance.
The JDBC URL is assembled separately:

```
def get_jdbc_url():
    return f"jdbc:db2://{DB2_HOSTNAME}:{DB2_PORT}/{DB2_DATABASE}"
```

- **Uploading CSV Files to Db2:** The core logic of the upload is in upload_csv_to_db2():
  1. Reads the CSV as a Spark DataFrame,
  2. Prints schema and sample rows,
  3. Uploads it to Db2 using .write.format("jdbc").

```python
def upload_csv_to_db2(spark_session, csv_file_name, db_table_name, limit_rows=None):
    file_path = os.path.join(SOURCE_DATA_DIR, csv_file_name)
    df = spark_session.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv(file_path)

    if limit_rows:
        df = df.limit(limit_rows)

    df.write.format("jdbc") \
        .option("url", get_jdbc_url()) \
        .option("dbtable", f"{DB2_SCHEMA}.{db_table_name}") \
        .options(**get_jdbc_properties()) \
        .mode("overwrite") \
        .save()
```

Step 5 : Load Cleaned Source Data and Generate Order Summary, Then Write to Db2

- Load the cleaned source CSV files into Spark DataFrames. These include product data, order items, orders, users, inventory items, and distribution centers.
- Perform multiple joins across these DataFrames to enrich the order details. This includes linking order information with user details, product data, inventory items, and distribution center information.
- Select and assemble relevant columns to create a comprehensive order summary. This summary includes order IDs, user details (name, email, city), product details, quantities, prices, and distribution center names.
- Write the enriched order summary DataFrame into the Db2 database table named ORDER_SUMMARY.
- Use Spark's JDBC capabilities with overwrite mode to ensure the database table is refreshed with the latest data.

```python
def process_and_upload_order_summary(spark_session, file_limits=None):

    print("\n--- Processing and Uploading ORDER_SUMMARY to DB2 ---")

    required_csv_details = {
        "order_items_df": {"file_name": "cleaned_order_items.csv"},
        "products_df": {"file_name": "cleaned_products.csv"},
        "inventory_df": {"file_name": "cleaned_inventory_items.csv"}
    }

    loaded_dfs = {}
    missing_file_found = False

    try:

        for alias, details in required_csv_details.items():
            file_name = details["file_name"]
            file_path = os.path.join(SOURCE_DATA_DIR, file_name)

            if not os.path.exists(file_path):
                print(f"ERROR: Required file for Order Summary not found: {file_path}. Skipping this operation.")
                missing_file_found = True
                break

            print(f"Reading {file_name} as '{alias}'...")
            temp_df = spark_session.read \
                .option("header", "true") \
                .option("inferSchema", "true") \
                .csv(file_path)

            limit = file_limits.get(file_name) if file_limits else None
            if limit is not None:
                print(f"Limiting {file_name} to first {limit} rows.")
                temp_df = temp_df.limit(limit)

            loaded_dfs[alias] = temp_df.alias(alias)
```

```python
        if missing_file_found:
            print("ERROR: Missing required input files. Aborting ORDER_SUMMARY generation.")
            return

        order_items_df = loaded_dfs.get("order_items_df")
        products_df = loaded_dfs.get("products_df")
        inventory_df = loaded_dfs.get("inventory_df")

        if not all([order_items_df, products_df, inventory_df]):
            print("ERROR: Not all required DataFrames could be loaded. Aborting ORDER_SUMMARY ETL.")
            return

        joined_df = order_items_df.join(
            inventory_df,
            order_items_df["inventory_item_id"] == inventory_df["id"],
            "left"
        )

        joined_df = joined_df.join(
            products_df,
            col("inventory_df.product_id") == products_df["id"],
        )

        order_summary_df = joined_df.groupBy(order_items_df["order_id"]).agg(
            count("*").alias("total_items"),
            _sum(order_items_df["sale_price"]).alias("total_amount"),
            countDistinct(col("inventory_df.product_id")).alias("unique_products"),
            _min(order_items_df["created_at"]).alias("first_order_date"),
            _max(order_items_df["created_at"]).alias("last_order_date")
        ).orderBy("order_id")

        print("Schema for ORDER_SUMMARY:")
        order_summary_df.printSchema()
        print("First 5 rows of ORDER_SUMMARY:")
        order_summary_df.show(5)
```

Step 6 : Calculate Most Sold Products by Aggregation and Write to Db2

- Load the cleaned order items, inventory items, products, and distribution center CSV files into Spark DataFrames.
- Join these DataFrames to connect sales data with product and distribution center information.
- Aggregate the total quantity sold for each product per distribution center by grouping on product ID, product name, and distribution center.
- Sort the results in descending order based on total quantities sold to identify top-selling products.
- Write the aggregated results into the Db2 table named MOST_SOLD_PRODUCTS.
- Use Spark JDBC with overwrite mode to update the Db2 table with the latest aggregation results.

```
def process_and_upload_most_sold_products(spark_session, file_limits=None):
    print("\n--- Processing and Uploading MOST_SOLD_PRODUCTS to DB2 ---")

    required_csv_files = {
        "order_items": "cleaned_order_items.csv",
        "products": "cleaned_products.csv",
        "distribution": "cleaned_distribution_centers.csv",
        "inventory": "cleaned_inventory_items.csv"
    }

    order_items_df = None
    products_df = None
    distribution_df = None
    inventory_df = None

    try:
        for df_name, file_name in required_csv_files.items():
            file_path = os.path.join(SOURCE_DATA_DIR, file_name)

            if not os.path.exists(file_path):
                print(f"ERROR: Required file for Most Sold Products not found: {file_path}. Skipping this operation.")
                return

            print(f"Reading {file_name}...")
            temp_df = spark_session.read \
                .option("header", "true") \
                .option("inferSchema", "true") \
                .csv(file_path)

            limit = file_limits.get(file_name) if file_limits else None
            if limit is not None:
                print(f"Limiting {file_name} to first {limit} rows.")
                temp_df = temp_df.limit(limit)

            if df_name == "order_items":
                order_items_df = temp_df
            elif df_name == "products":
```

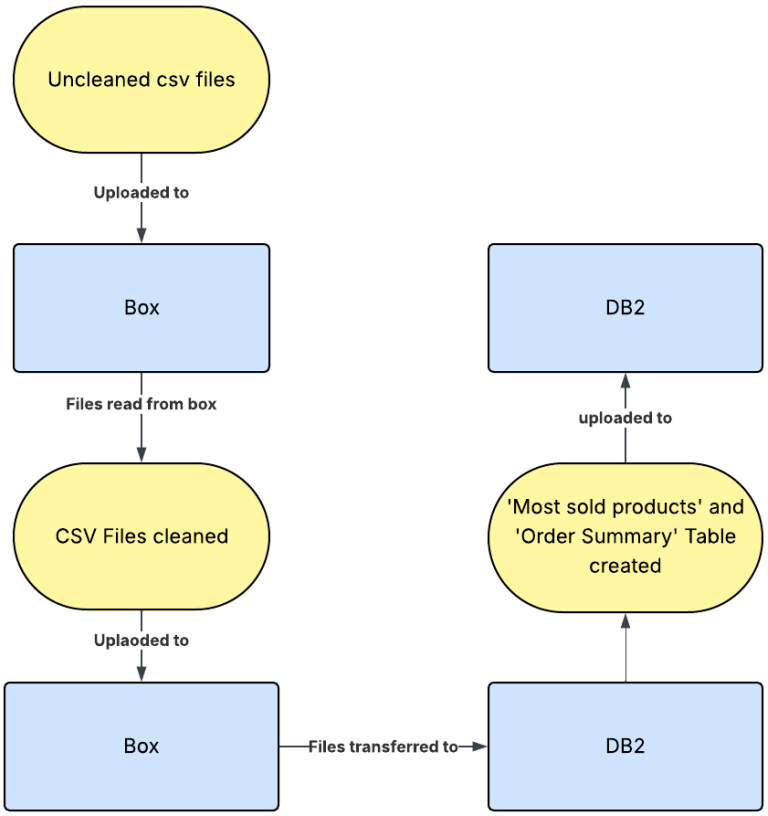
```
            temp_df = temp_df.limit(limit)

        if df_name == "order_items":
            order_items_df = temp_df
        elif df_name == "products":
            products_df = temp_df
        elif df_name == "distribution":
            distribution_df = temp_df
        elif df_name == "inventory":
            inventory_df = temp_df

    if not all([order_items_df, products_df, distribution_df, inventory_df]):
        print("ERROR: Not all required DataFrames could be loaded. Aborting Most Sold Products analysis.")
        return

    most_sold_products = order_items_df \
        .join(inventory_df, order_items_df["inventory_item_id"] == inventory_df["id"], "left") \
        .join(products_df, inventory_df["product_id"] == products_df["id"], "left") \
        .join(distribution_df, inventory_df["product_distribution_center_id"] == distribution_df["id"], "left") \
        .groupBy(
            products_df["id"].alias("product_id"),
            products_df["name"].alias("product_name"),
            distribution_df["name"].alias("distribution_center")
        ) \
        .agg(
            _sum(lit(1)).alias("total_quantity_sold")
        ) \
        .orderBy("total_quantity_sold", ascending=False)

    print("Schema for MOST_SOLD_PRODUCTS:")
    most_sold_products.printSchema()
    print("First 5 rows of MOST_SOLD_PRODUCTS:")
    most_sold_products.show(5)
```

Step 7 : Run the Full Pipeline Script

- Use a PowerShell script to automate the pipeline execution.
- The script sets the working directory to its own location.
- It activates the Python virtual environment to ensure dependencies and environment variables are properly loaded.
- Sequentially run the following Python scripts:
    - box_operations.py for Box file operations.
    - db2_data_upload.py to upload cleaned CSV source data to Db2 tables.
    - most_sold_products.py to calculate and upload the most sold products aggregation.
    - order_summary.py to generate and upload the enriched order summary.
- The script prints progress and completion messages for tracking.
- This automation ensures the entire data pipeline runs smoothly in the correct order with consistent environment settings.

```powershell
$scriptDir = Split-Path -Parent $MyInvocation.MyCommand.Definition
Set-Location -Path $scriptDir

$venvPython = Join-Path $scriptDir ".\.venv\Scripts\python.exe"

if (-not (Test-Path $venvPython)) {
    Write-Error "Virtual environment Python executable not found: $venvPython. Please ensure .venv is created and activated at least once."
    exit 1
}

Write-Host "Running pipeline using Python from virtual environment: $venvPython"

$boxIntegrationScript = Join-Path $scriptDir "scripts\box_operations.py"
$db2UploadScript = Join-Path $scriptDir "scripts\db2_data_upload.py"
$mostSoldScript = Join-Path $scriptDir "scripts\most_sold_products.py"
$orderSummaryScript = Join-Path $scriptDir "scripts\order_summary.py"

& $venvPython $boxIntegrationScript

& $venvPython $db2UploadScript

& $venvPython $mostSoldScript

& $venvPython $orderSummaryScript

Write-Host "Pipeline execution complete."
```

Data flow Diagram



Conclusion

The pipeline efficiently integrates and transforms data from multiple sources — orders, users, products, inventory, and distribution centers — into meaningful, aggregated insights using PySpark. By joining these datasets, selecting relevant fields, and performing group-wise aggregations, the workflow identifies top-selling products across different distribution centers. This structured approach not only ensures data completeness through careful join strategies but also leverages Spark's distributed processing power to handle large-scale data efficiently. The final summarized data is then seamlessly written back to an IBM Db2 database, enabling downstream analytics, reporting, and business decision-making based on accurate, consolidated sales information.