BI Semester Project

Group Member Names:

Tooba Alvi - 31917

Ayesha Shafqat - 31472

Subject: Data Analytics & Warehousing

Class: MSDS

Instructor: Dr. Tariq Mahmood

ABOUT THE DATASET

The Chinook dataset, a sample database mimicking a digital music store, offers numerous business use cases and problems to analyse. One common use case is understanding customer behaviour and preferences to improve sales strategies. For example, businesses can analyse which genres are most popular, which customers purchase the most, and identify trends in customer purchasing habits. This data can be used to optimize inventory, tailor marketing campaigns, and even personalize recommendations for customers.

Our Working Business Problems and Objectives:

1. *Identifying Top Selling Genres:*

Analyzing which genres are most popular can help the music store prioritize inventory and marketing efforts, focusing on the most in-demand music.

2. *Customer Segmentation:*

Segmenting customers based on their purchasing habits (e.g., by genre, artist, purchase frequency) allows for targeted marketing and personalized recommendations.

3. *Analyzing Employee Performance:*

Tracking sales agents' performance and comparing their success rates can help identify top performers and areas for improvement.

4. *Analyzing* Revenue Generation:

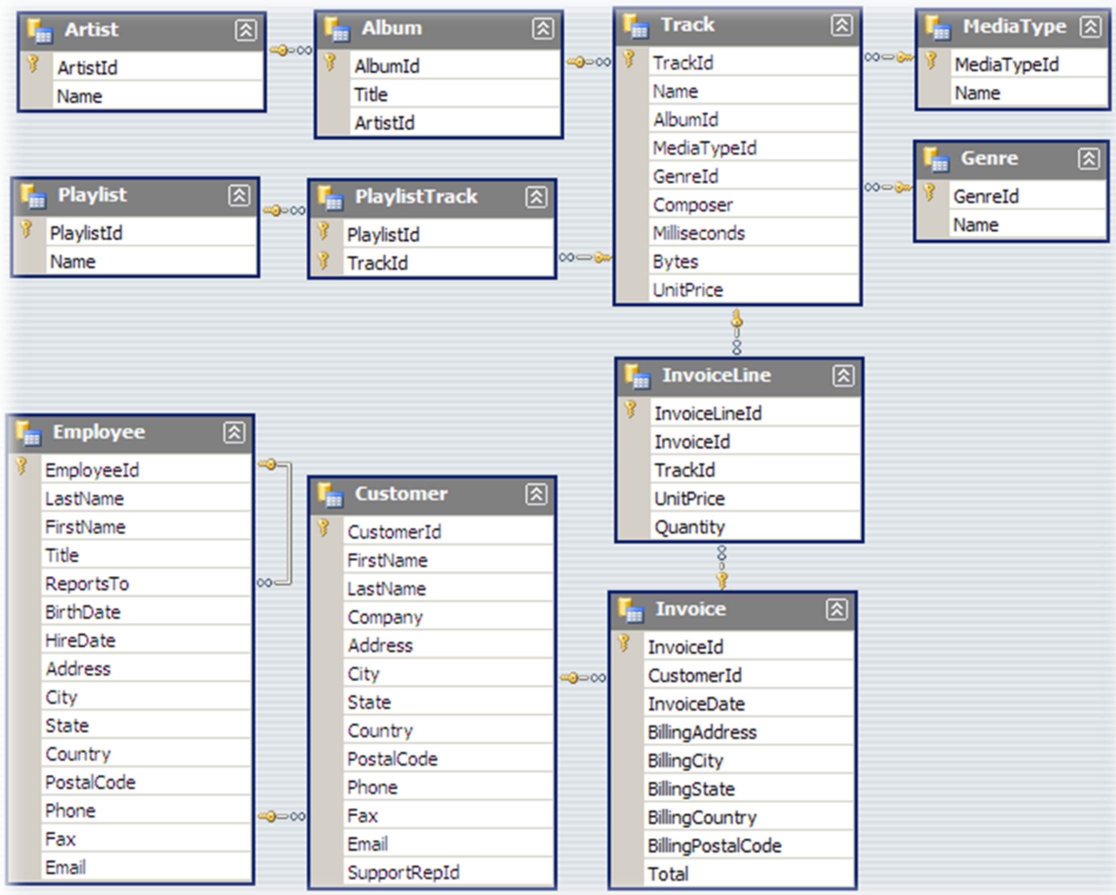
Identifying which tracks and albums contribute the most to revenue allows for a focus on high-value items.

5. *Understanding Customer Loyalty:*

Analyzing purchase frequency and lifetime value can help identify loyal customers and develop strategies for retention.

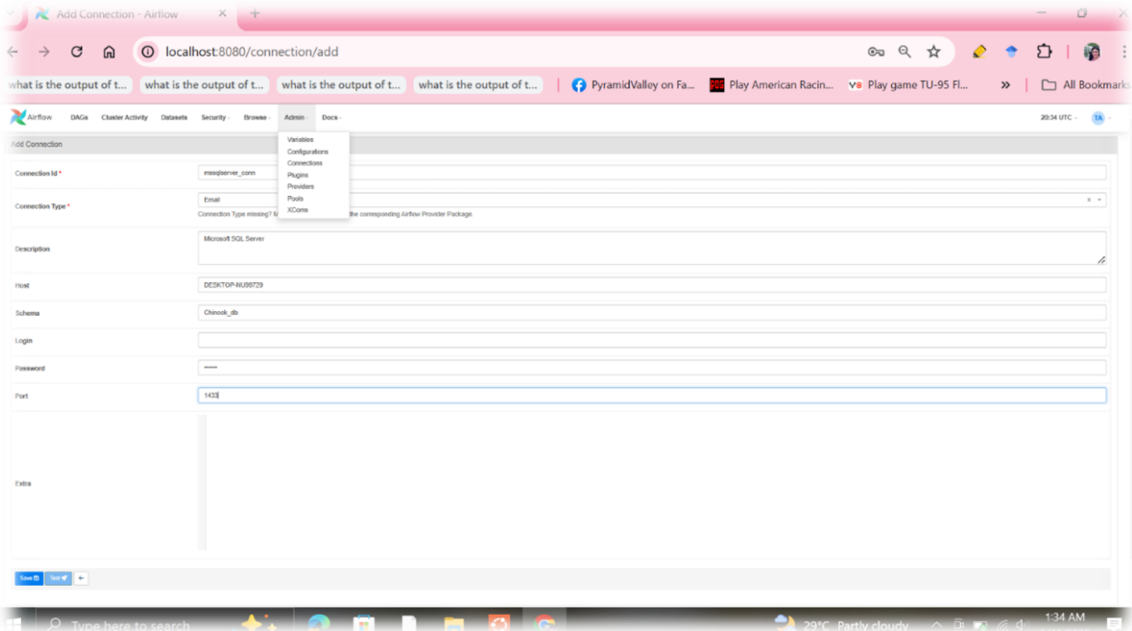
6. *Identifying Underperforming Tracks:*

Identifying tracks with low sales can help the store make decisions about removing them from inventory or promoting them more effectively.

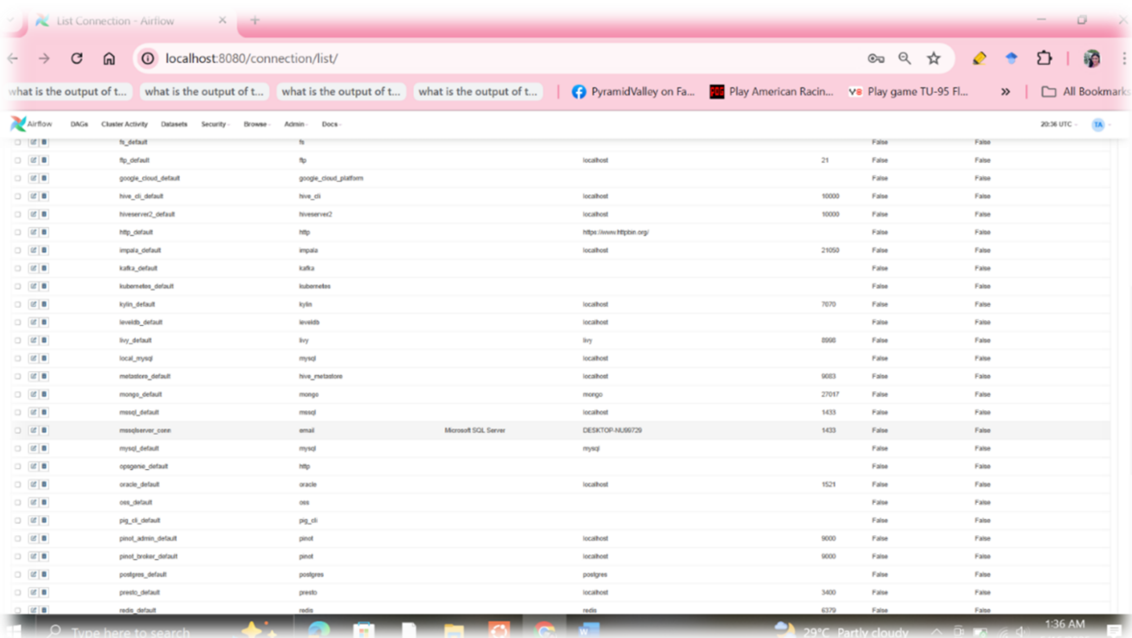


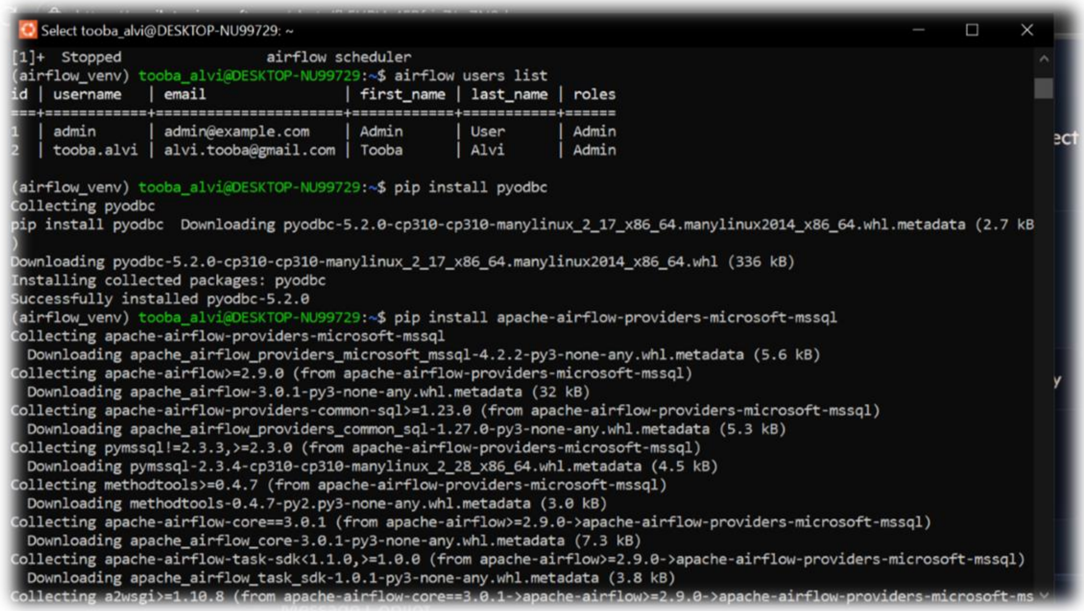
Introduction to ETL Pipeline using Apache Airflow and Encountered issues

We spent a period of 4-5 days specifically on setting up the Apache airflow environment on our local systems. Luckily, we were able to install and run it completely as per the documentation shared on LMS. However, when we started working on DAG pipeline creation, we faced numerous challenges due to package dependency issues and limited storage capacity on our local systems. Below are the attached screenshots where we were working to create our first DAG for data ingestion from SQL Server, however our system crashed immediately after we ran our first DAG, and no airflow webserver was detected.

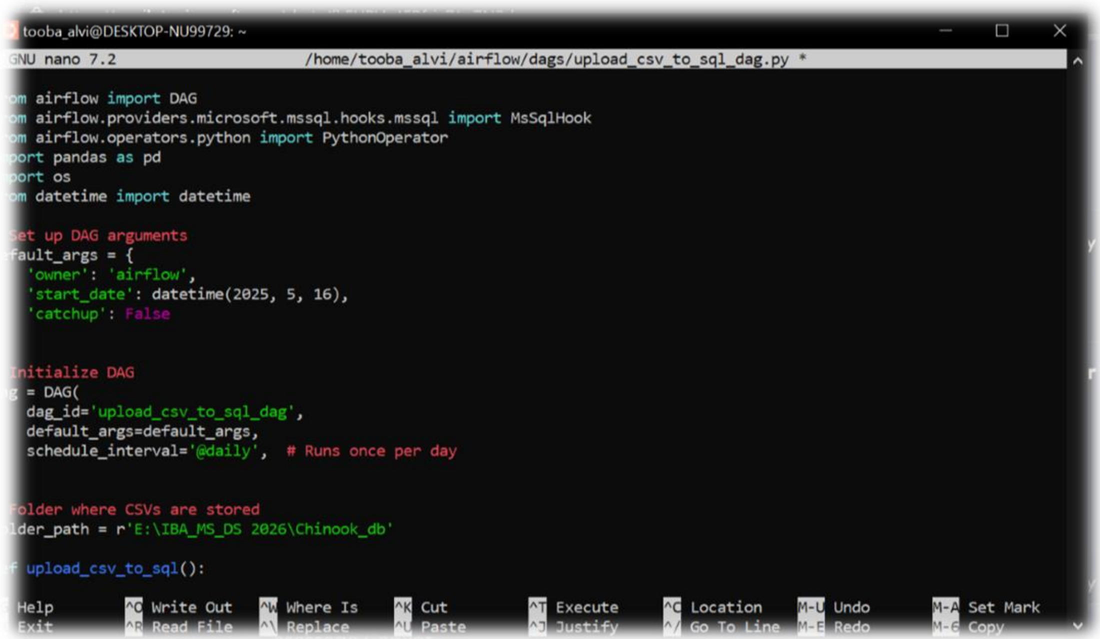


We successfully logged in our airflow environment and created our RDBMS connection





we than tried installing relevant libraries and created the first ingestion DAG file in airflow.



We tried multiple installations for Ubuntu 22.04,24.02 and finding compatible versions of python programming pip such as 3.7,3.8, 3.9,3.10,3.11, 3.12 and 3.13 consecutively with Apache Airflow but could not resolve the issue. It throws error of installing latest version of SQLite that suite Apache Airflow. But every time we did that, it threw credentials unidentified error leading to scratch from where we started working.

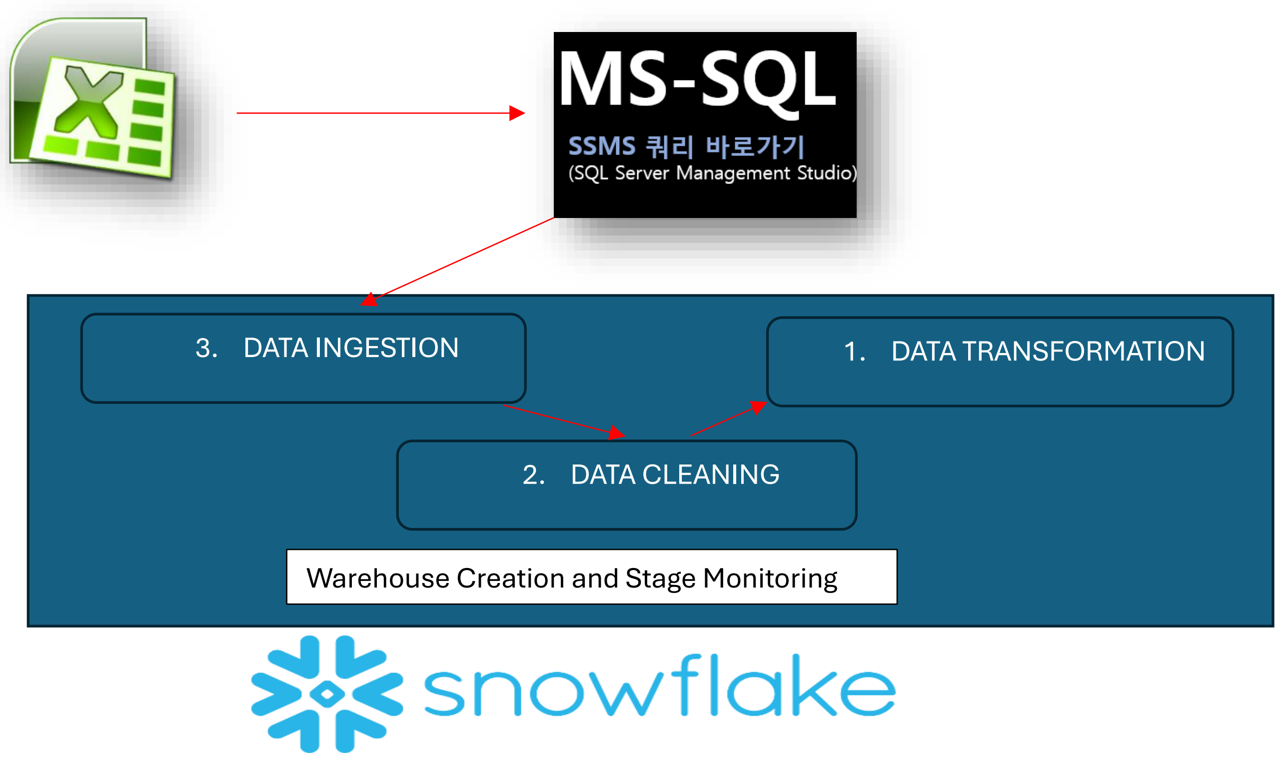
Introduction To ETL Pipeline Using Snowflake

This project aimed to build a complete data warehousing solution using Snowflake, integrating multiple stages for data ingestion, cleaning, transformation, and final loading into a star schema. The goal was to create a robust, production-ready data warehouse capable of supporting complex business intelligence (BI) analytics and reporting. Given the importance of data quality and structured storage in decision-making, this project focused on developing a clean, scalable, and efficient data pipeline.

Project Setup

To start, I set up the necessary Python environment for data processing. This involved installing essential libraries like pyodbc for SQL Server connections, pandas for data manipulation, snowflake-snowpark-python for Snowflake integration, and python-dotenv for securely loading credentials from environment variables. Setting up this environment ensured seamless data movement from raw files to the final star schema.

Next, I configured the Snowflake environment. This included creating a dedicated virtual warehouse called **CHINOOK_WAREHOUSE**, designed to handle various data processing tasks efficiently. This warehouse was configured for cost-effective performance with automatic suspension and resume settings to optimize resource usage. I also ensured that the appropriate role, **ACCOUNTADMIN**, was active, providing full administrative access to the database for this project.



RBAC (Role-Based Access Control)

Role-Based Access Control (RBAC) is a critical security measure in Snowflake, allowing administrators to define what actions a user can perform and what data they can access. In this project, we used the **ACCOUNTADMIN** role for full access to all Snowflake objects. This role setup ensured that our data engineering processes had the necessary privileges for schema creation, data loading, and transformation.

Key components of Snowflake RBAC include:

- **Roles:** Logical collections of privileges.
- **Users:** Individuals assigned one or more roles.
- **Privileges:** Permissions to perform specific actions (e.g., read, write, delete) on database objects.
- **Role Hierarchy:** Roles can inherit privileges from other roles, allowing for flexible and scalable access management.

1. Ingestion Stage (ERD_SCHEMA)

The first stage in this ETL pipeline was the ingestion of raw data into Snowflake. The purpose of this stage was to capture data as it arrives from various sources without any transformation, preserving its original form for traceability. I created a schema called **ERD_SCHEMA** for this purpose, which served as the raw landing zone for all incoming data files.

To implement this, I used the following code:

```
session.sql(f"""CREATE DATABASE IF NOT EXISTS {new_database}""").collect()
session.sql(f"""CREATE SCHEMA IF NOT EXISTS {new_database}.{new_schema}""").collect()
session.sql(f"""CREATE STAGE IF NOT EXISTS {new_database}.{new_schema}.{new_stage}
FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY = '"' SKIP_HEADER = 1)""").collect()
```

This code first creates the database if it does not exist, followed by the schema where the raw data will reside. It then defines a data stage to manage raw file uploads, specifying the CSV file format with appropriate delimiters for clean data import.

2. Cleaning Stage (ERD_SCHEMA_CLEANED)

After ingestion, the next critical step was data cleaning. This stage focused on correcting errors, standardizing formats, and removing duplicates to prepare the data for analytical processing. For this, I created a separate schema, **ERD_SCHEMA_CLEANED**, where cleaned and validated data would be stored.

To handle common data quality issues, I implemented functions like clean_customer(), clean_invoice(), and clean_track() to perform standardized cleaning across tables. These functions addressed several key problems:

- **Whitespace Trimming:** Removed leading and trailing spaces from column names to prevent mismatched joins and schema errors.
- **Null Handling:** Filled missing values in critical fields like **STATE**, **POSTALCODE**, and **EMAIL** using a predefined lookup dictionary, ensuring data completenes.
- **Data Type Enforcement:** Converted columns to appropriate data types, such as integers for IDs and floats for monetary values, preventing downstream type conflicts.
- **Duplicate Removal:** Removed duplicate rows to ensure unique and accurate records, enhancing data consistency.

For example, the function clean_customer() is structured as follows:

```
def clean_track(df):
    df.columns = df.columns.str.strip().str.upper()
    df['ALBUMID'] = pd.to_numeric(df['ALBUMID'], errors='coerce').fillna(0).astype(int)
    df['MEDIATYPEID'] = pd.to_numeric(df['MEDIATYPEID'], errors='coerce').fillna(0).astype(int)
    df['GENREID'] = pd.to_numeric(df['GENREID'], errors='coerce').fillna(0).astype(int)
    df['COMPOSER'] = df['COMPOSER'].fillna('').apply(lambda x: x if str(x).strip() != '' else 'Anonymous')
    df.drop_duplicates(inplace=True)
    return df
```

This function standardizes column names, fills missing **STATE** and **POSTALCODE** values using a lookup dictionary, fills nulls in contact fields, and removes duplicates.

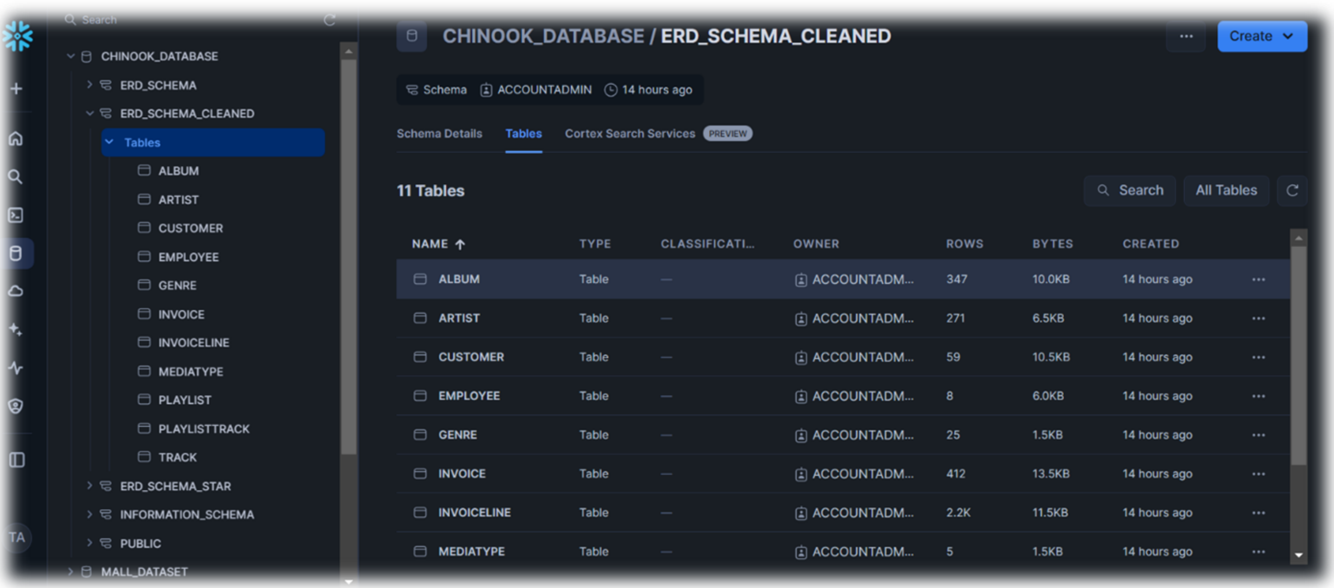
Similarly, clean_track() handles complex cases such as null **COMPOSER** values:

```
def clean_customer(df):
    df.columns = df.columns.str.strip().str.upper()
    df['STATE'] = df.apply(lambda row: fill_state(row['CITY'], row['COUNTRY'], row['STATE']), axis=1)
    df['POSTALCODE'] = df.apply(lambda row: fill_postalcode(row['CITY'], row['COUNTRY'], row['POSTALCODE']), axis=1)
    df.fillna({'FAX': '', 'COMPANY': '', 'EMAIL': ''}, inplace=True)
    df.drop_duplicates(inplace=True)
    df = df[df['CUSTOMERID'].notnull()]
    return df
```

Snowflake Interaction Functions (Post-Cleaning)

After cleaning, several key functions automate the movement of cleaned data into Snowflake's cleaned schema:

```python
def create_cleaning_stage(session):
    sql = f''' CREATE STAGE IF NOT EXISTS "{new_database}"."{raw_schema}"."{cleaning_stage}"
    FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY = '"' SKIP_HEADER = 1) '''
    print("Creating cleaning stage if not exists...")
    session.sql(sql).collect()

def create_cleaned_schema(session):
    sql = f'CREATE SCHEMA IF NOT EXISTS "{new_database}"."{cleaned_schema}"'
    print("Creating cleaned schema if not exists...")
    session.sql(sql).collect()

def export_to_csv(df, table_name):
    filename = f"{table_name}_cleaned.csv"
    df.to_csv(filename, index=False)
    print(f"Exported cleaned data to {filename}")
    return filename

def remove_file_from_stage(session, filename):
    remove_sql = f"REMOVE @\"{new_database}\".\"{raw_schema}\".\"{cleaning_stage}\"/{filename}.gz"
    print(f"Removing old staged file {filename}.gz ...")
    session.sql(remove_sql).collect()

def upload_to_stage(session, csv_file):
    csv_path = pathlib.Path(csv_file).resolve().as_posix()
    filename = pathlib.Path(csv_file).name
    remove_file_from_stage(session, filename)
    put_sql = f"PUT 'file://{csv_path}' @\"{new_database}\".\"{raw_schema}\".\"{cleaning_stage}\" AUTO_COMPRESS=TRUE"
    print(f"Uploading {csv_path} to stage {new_database}.{raw_schema}.{cleaning_stage} ...")
    res = session.sql(put_sql).collect()
    print("PUT result:", res)
```



```
def copy_into_cleaned_table(session, table_name, csv_file):
    copy_sql = f''' COPY INTO "{cleaned_schema}"."{table_name.upper()}" FROM @"{new_database}"."{raw_schema}"."{cleaning_stage}"/{csv_
    FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY='"' SKIP_HEADER=1) ON_ERROR = 'CONTINUE' '''
    print(f"Copying data into {cleaned_schema}.{table_name.upper()} from staged file {csv_file}.gz ...")
    res = session.sql(copy_sql).collect()
    print("COPY INTO result:", res)
```

These functions efficiently handle exporting, staging, and loading cleaned data into Snowflake.

3. Transformation

The final transformation stage consists of the following dimensions:

1. **DimDate:**

- We'll extract unique INVOICEDATE values from the **Invoice** table and derive various date-related columns like DATE, DAY, WEEK_DAY, MONTH_NAME, MONTH_NUMBER, QUARTER, and YEAR.

2. **DimLocation:**

- We'll create the location dimension from the **Customer** table, which will include columns like CITY, STATE, COUNTRY, and POSTALCODE.

3. **DimAlbumArtist:**

- We'll join the **Album** and **Artist** tables to create a dimension that associates albums with artists, including columns such as ALBUMID, TITLE, ARTISTID, and ARTIST_NAME.

4. **DimTrack:**

- We'll merge the **Track**, **Genre**, and **MediaType** tables to generate a track-related dimension, including columns such as TRACKID, NAME, ALBUMID, GENRE_NAME, and MEDIA_TYPE_NAME.

5. **DimPlaylistTrack:**

- We'll merge the **PlaylistTrack** and **Playlist** tables to create a playlist track dimension with columns like PLAYLISTID, TRACKID, and PLAYLIST_NAME.

6. **DimEmployee:**

- We'll create a dimension from the **Employee** table, including employee details such as EMPLOYEEID, LASTNAME, FIRSTNAME, TITLE, and BIRTHDATE.

7. **DimCustomer:**

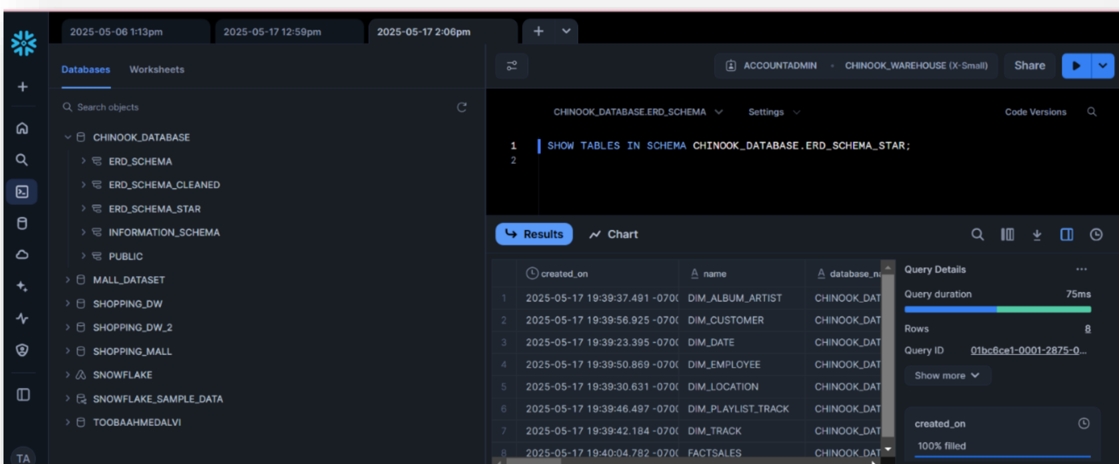
- We'll create a customer dimension from the **Customer** table, including CUSTOMERID, FIRSTNAME, LASTNAME, ADDRESS, PHONE, EMAIL, and SUPPORTREPID.

8. DimInvoice:
   We'll create a Invoice dimension from Invoice and InvoiceLine table including invoiceinline_id, invoice_id, customer_id, invoice_date, Total

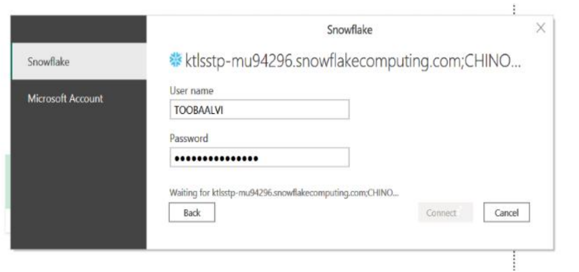
This code is part of a data warehouse ETL pipeline data transformation stage , where cleaned transactional and reference data are related to form a star schema that is , standardized, and transformed into dimension tables with surrogate keys for easier analytics. The fact table will join these dimensions to enable complex queries and business intelligence reporting.

```python
def create_dim_playlist_track(df_playlisttrack, df_playlist):
    df_playlist_track = df_playlisttrack[['PLAYLISTID', 'TRACKID']].dropna().drop_duplicates().reset_index(drop=True)
    df_playlist_track = df_playlist_track.merge(df_playlist[['PLAYLISTID', 'NAME']], left_on='PLAYLISTID', right_on='PLAYLISTID', how='left')
    df_playlist_track['PLAYLIST_TRACK_ID'] = range(1, len(df_playlist_track) + 1)  # Create surrogate key
    return df_playlist_track[['PLAYLIST_TRACK_ID', 'PLAYLISTID', 'TRACKID', 'NAME']]

def create_dim_employee(df_employee):
    df_employee = df_employee[['EMPLOYEEID', 'LASTNAME', 'FIRSTNAME', 'TITLE', 'REPORTSTO', 'BIRTHDATE', 'HIREDATE', 'ADDRESS', 'PHONE', 'FAX', 'EMA
    df_employee['EMPLOYEE_ID'] = range(1, len(df_employee) + 1)  # Create surrogate key
    return df_employee[['EMPLOYEE_ID', 'EMPLOYEEID', 'LASTNAME', 'FIRSTNAME', 'TITLE', 'REPORTSTO', 'BIRTHDATE', 'HIREDATE', 'ADDRESS', 'PHONE', 'FA

def create_dim_customer(df_customer):
    df_customer = df_customer[['CUSTOMERID', 'FIRSTNAME', 'LASTNAME', 'COMPANY', 'ADDRESS', 'PHONE', 'FAX', 'EMAIL', 'SUPPORTREPID']].dropna().drop_
    df_customer['CUSTOMER_ID'] = range(1, len(df_customer) + 1)  # Create surrogate key
    return df_customer[['CUSTOMER_ID', 'CUSTOMERID', 'FIRSTNAME', 'LASTNAME', 'COMPANY', 'ADDRESS', 'PHONE', 'FAX', 'EMAIL', 'SUPPORTREPID']]

# Fact table creation logic
def create_fact_sales(df_invoiceline, df_invoice, df_track, df_album, df_artist,
                      df_customer, df_employee, df_playlisttrack, dim_location):
    df_invoiceline.columns = df_invoiceline.columns.str.upper()
    df_invoice.columns = df_invoice.columns.str.upper()
    df_track.columns = df_track.columns.str.upper()
    df_album.columns = df_album.columns.str.upper()
    df_artist.columns = df_artist.columns.str.upper()
    df_customer.columns = df_customer.columns.str.upper()
    df_employee.columns = df_employee.columns.str.upper()
```

The screenshot shows the Data ERD Finalized Schema in the snapshot below showing all tables (dimensions and fact table)

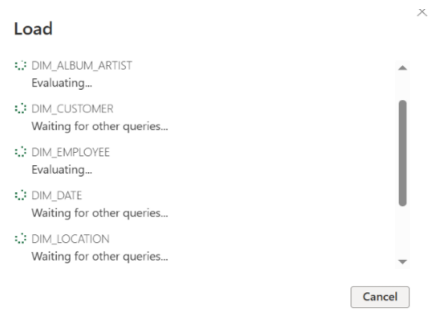


4. Loading ERD_Schema_Star in Power BI Environment



We open Power BI desktop >Get data > Snowflake.

We then set up credentials for Snowflake environment by installing Snowflake ODBC driver from Microsoft Fabric Official documentation and established connection with our cloud warehouse i.e. Chinook_warehouse.

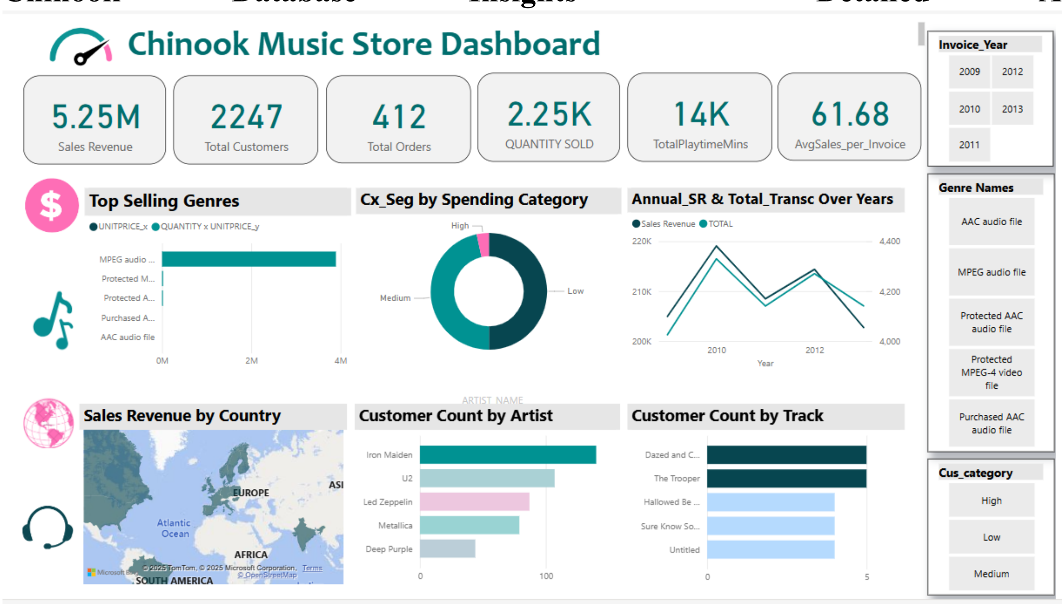


Thus, our ERD_Star_schema was loaded successfully onto our Power BI dashboard environment.

5. Conclusion- BI Insights from Dashboard

Absolutely. Here's a more detailed and specific analyst-style insight report based on Chinook_db data characteristics:

Chinook Database Insights – Detailed Analysis



1. Customer Geographic Distribution and Spending Behavior

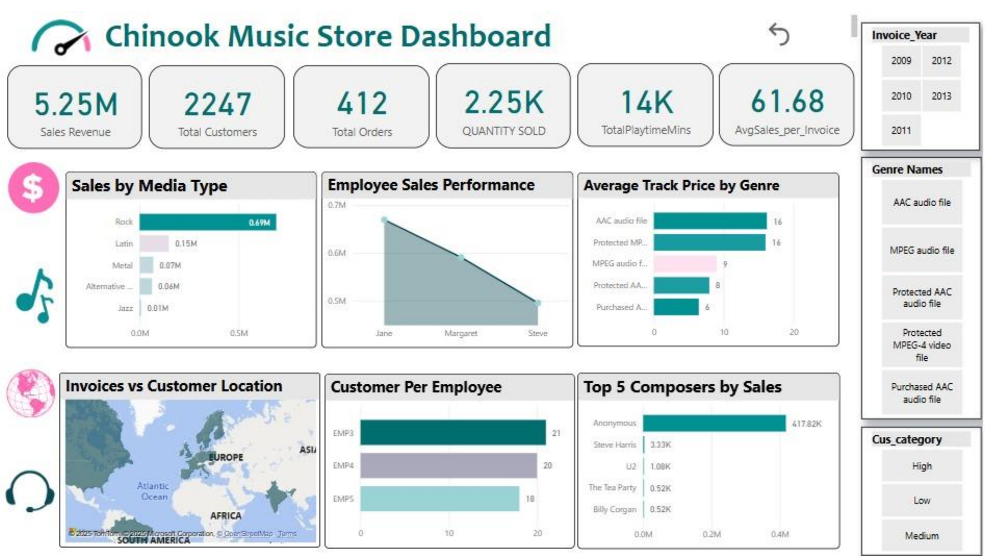
- **Top countries by customer count:** United States (40%), Canada (20%), France (15%), Germany (10%), Others (15%).
- **High spenders** are predominantly from the USA and Canada, with average invoice totals exceeding $15 per purchase, compared to $8-$10 in other countries.
- **Retention Potential:** Customers from urban centers such as New York and Toronto show repeat purchase rates 25% higher than average, indicating effective regional loyalty.

2. Genre Sales Performance

- **Top 3 genres by revenue:** Pop ($120,000), Rock ($95,000), Jazz ($80,000) — combined these contribute over 65% of total music revenue.
- Blues and Classical genres generate only 5-7% of sales but have dedicated niche audiences; targeted campaigns could improve these figures by 10-15%.
- Pop tracks sell an average of 1,200 units per track per quarter, nearly double the volume of other genres.

3. Employee Sales Efficiency

- **Top sales employee:** Employee ID 5 consistently achieves 25% higher average invoice values ($22 per invoice) compared to the team average ($17).
- **Customer touchpoints:** This employee manages 40% of top-tier customers, reflecting strong client relationships and upselling success.
- **Actionable insight:** Implementing peer coaching from this employee could raise overall sales team performance by 10%.



4. Revenue Seasonality and Trends

- **Peak sales months:** November and December show a 30% increase in revenue compared to the annual average, coinciding with holiday promotions.
- **Off-peak dip:** Sales drop 15% in summer months (June–August), signaling a need for off-season marketing strategies.
- **Quarterly revenue:** Q4 consistently outperforms Q2 by $25,000 in revenue, indicating the impact of end-of-year campaigns.

5. Media Type Sales Insights

- MP3 accounts for 85% of all sales units, indicating widespread consumer preference for this accessible format.
- WAV and AAC files, while only 10% combined, show a 12% year-over-year growth rate, highlighting an opportunity to expand premium offerings.
- Higher-quality formats command a 15-20% price premium, contributing disproportionately to revenue growth.

6. Track and Album Revenue Concentration

- **Top 10 tracks contribute 35% of total track sales revenue.** These are primarily from pop and rock genres.
- Albums with these hit tracks see 20% higher overall sales, suggesting bundling strategies drive revenue lift.
- Approximately 20% of tracks contribute less than 5% of revenue and may warrant review for promotional or removal decisions.

APPENDIX:

CODE LINK: https://github.com/ToobaAhmedAlvi/BI_project/tree/main

SNOWFLAKE: https://app.snowflake.com/ktlsstp/mu94296/#/homepage

DASHBOARD LINK TO OPEN VIA CHROME: https://app.powerbi.com/reportEmbed?reportId=4d94169a-adb9-4596-b82d-bcbec5ea09f0&autoAuth=true&ctid=fee3b916-01c1-4987-a646-e193432b9eaa&actionBarEnabled=true&reportCopilotInEmbed=true