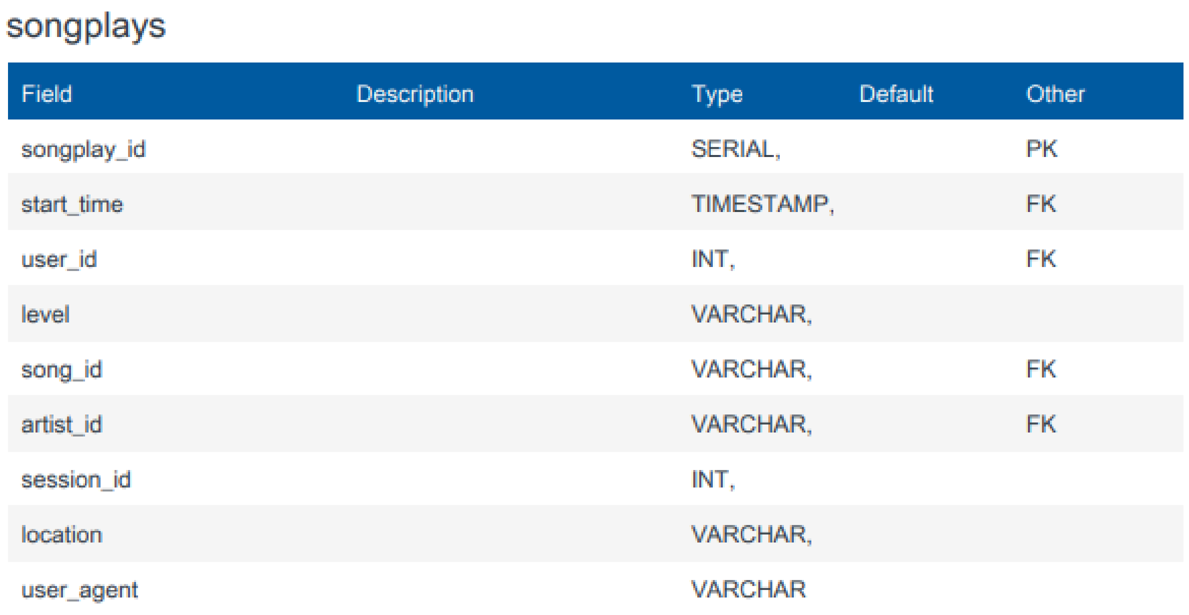
# DATA MODELLING WITH POSTGRESQL

1. **Data Overview**

- Song Dataset: files are partitioned by the first three letters of each song's track ID e.g. */data/song_data.*.json.
  Sample:
  {"artist_id": "ARD7TVE1187B99BFB1", "artist_latitude": null, "artist_location": "California - LA", "artist_longitude": null, "artist_name": "Casual", "duration": 218.93179, "num_songs": 1, "song_id": "SOMZWCG12A8C13C480", "title": "I Didn't Mean To", "year": 0}
- Log Dataset: files in the dataset you'll be working with are partitioned by year and month e.g. */data/log_data.*json.
  Sample:
  {"artist": "Stephen Lynch", "auth": "Logged In", "firstName": "Jayden", "gender": "M", "itemInSession": 0, "lastName": "Bell", "length": 182.85669, "level": "free", "location": "Dallas-Fort Worth-Arlington", "method": "TX PUT", "page": "NextSong", "registration": 1.540992.., "sessionId": "829", "song":"Jim Henson's Dead", "status": 200, "ts": 1543537327796, "userAgent": "Mozilla/5.0 (compatible; MSIE 10.0; Windows NT...", "userId": 91}

2. **What is Data Modelling?**
   Data modeling is a high level abstraction that organizes data and how they relate to each other.
3. **Schema for Song Play Analysis**
   Using the song and log datasets, we'll create a star schema optimized for queries on song play analysis.
   **Star Schema:** A star schema is the simplest style of data mart schema. The star schema consists of one or more fact tables referencing any number of dimension tables. It has some advantages like fast aggregation for analytics, simple queries for JOINs, etc.
   This includes the following tables:
   - **Facts Table**
     In data warehousing, a fact table consists of measurements, metrics or facts of a business process

1. songplays — records in log data associated with song plays, i.e., records with page NextSong . This filter for the page column specifies that the user has played a song, like clicked on the next song button in the app.
    - songplay_id, start_time, user_id, level, song_id, artist_id, session_id, location, user_agent

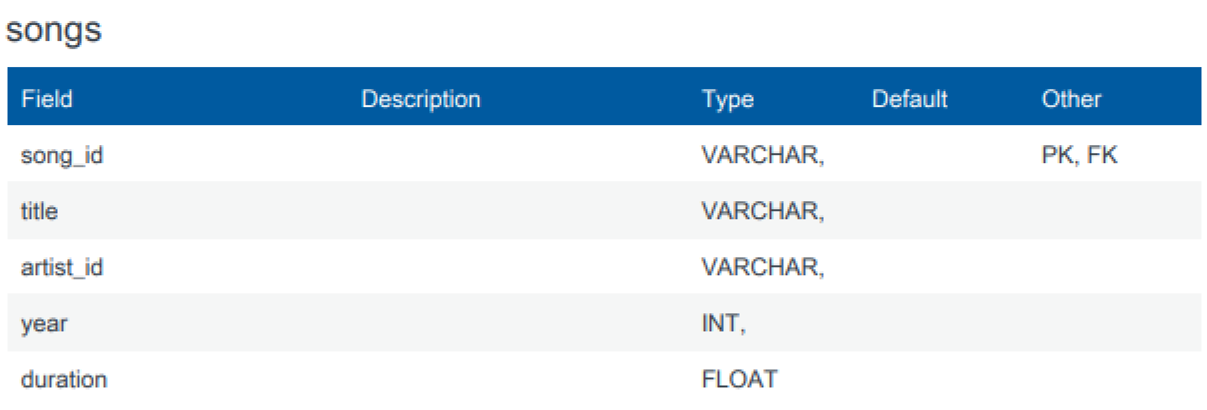


songplays

| Field | Description | Type | Default | Other |
|---|---|---|---|---|
| songplay_id | | SERIAL, | | PK |
| start_time | | TIMESTAMP, | | FK |
| user_id | | INT, | | FK |
| level | | VARCHAR, | | |
| song_id | | VARCHAR, | | FK |
| artist_id | | VARCHAR, | | FK |
| session_id | | INT, | | |
| location | | VARCHAR, | | |
| user_agent | | VARCHAR | | |

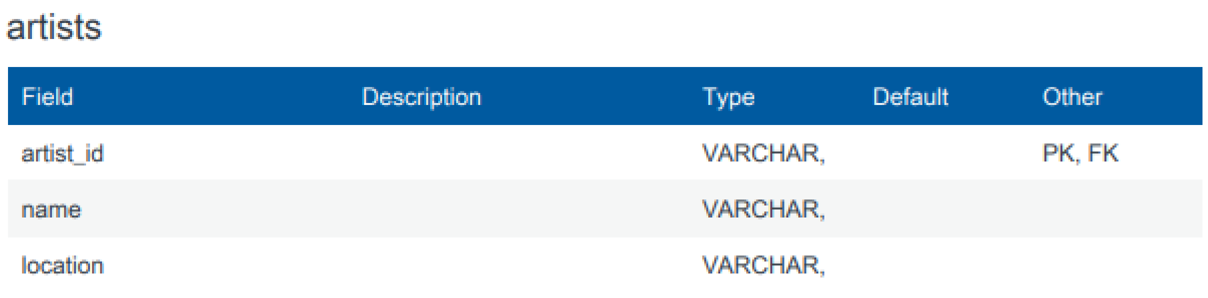
- **Dimension Tables**
  A dimension table is a structure that categorizes facts and measures in order to enable users to answer business questions. Commonly used dimensions are people, products, place and time.
  1. users — Following information about users:
      - user_id, first_name, last_name, gender, level

  2. songs — Following info about songs:
      - song_id, title, artist_id, year, duration

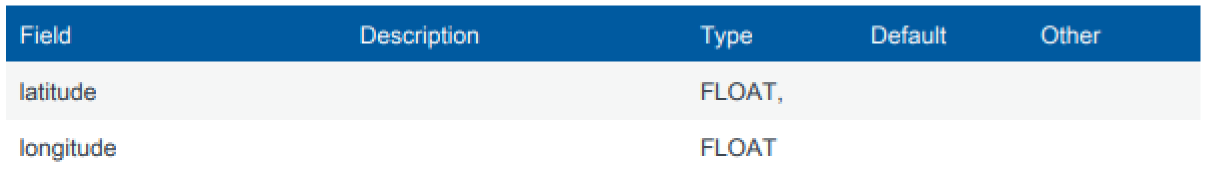

songs

| Field | Description | Type | Default | Other |
|---|---|---|---|---|
| song_id | | VARCHAR, | | PK, FK |
| title | | VARCHAR, | | |
| artist_id | | VARCHAR, | | |
| year | | INT, | | |
| duration | | FLOAT | | |

3. artists — Artists information:
   - artist_id, name, location, latitude, longitude

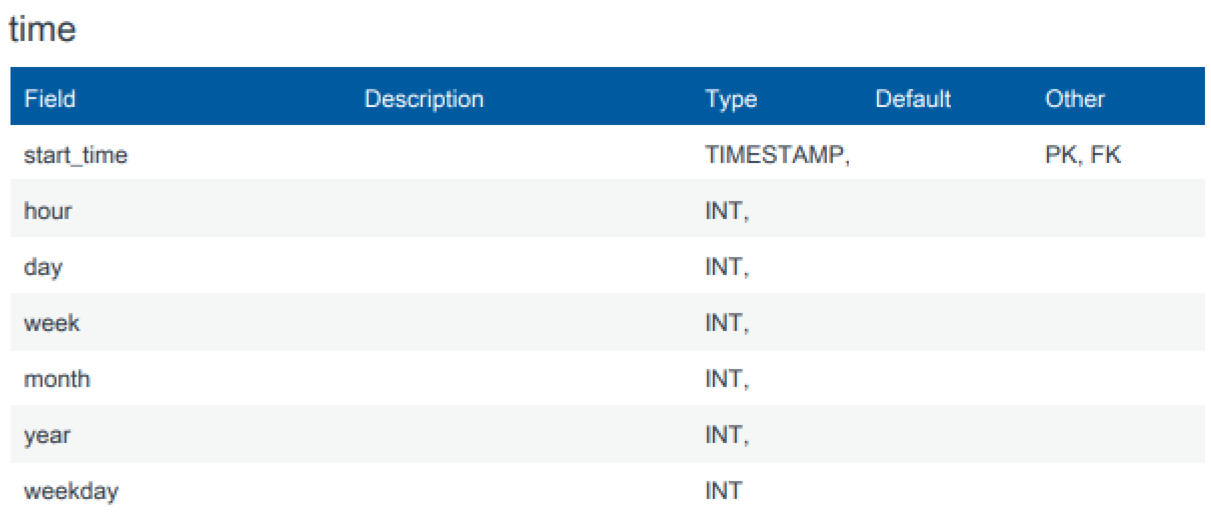

artists

| Field | Description | Type | Default | Other |
|---|---|---|---|---|
| artist_id | | VARCHAR, | | PK, FK |
| name | | VARCHAR, | | |
| location | | VARCHAR, | | |



| Field | Description | Type | Default | Other |
|---|---|---|---|---|
| latitude | | FLOAT, | | |
| longitude | | FLOAT | | |

4. time — Timestamp broken down into specific units:
   - start_time, hour, day, week, month, year, weekday

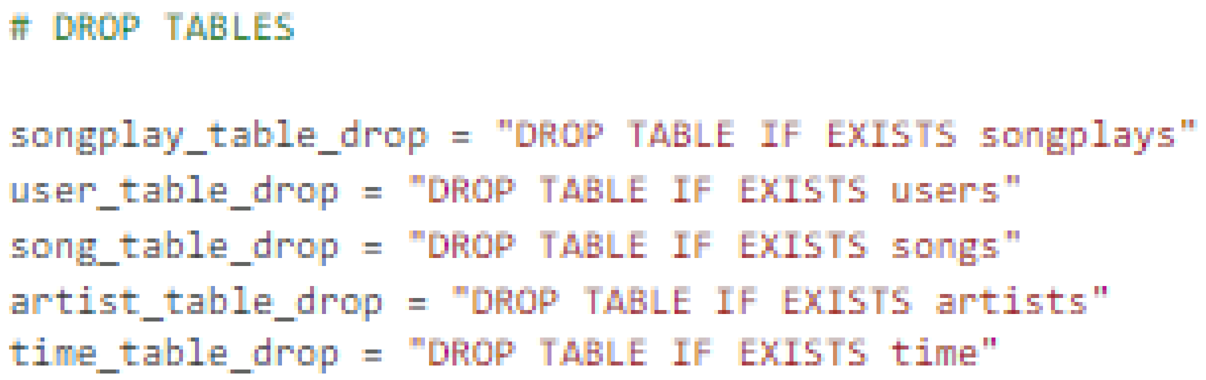

time

| Field | Description | Type | Default | Other |
|---|---|---|---|---|
| start_time | | TIMESTAMP, | | PK, FK |
| hour | | INT, | | |
| day | | INT, | | |
| week | | INT, | | |
| month | | INT, | | |
| year | | INT, | | |
| weekday | | INT | | |

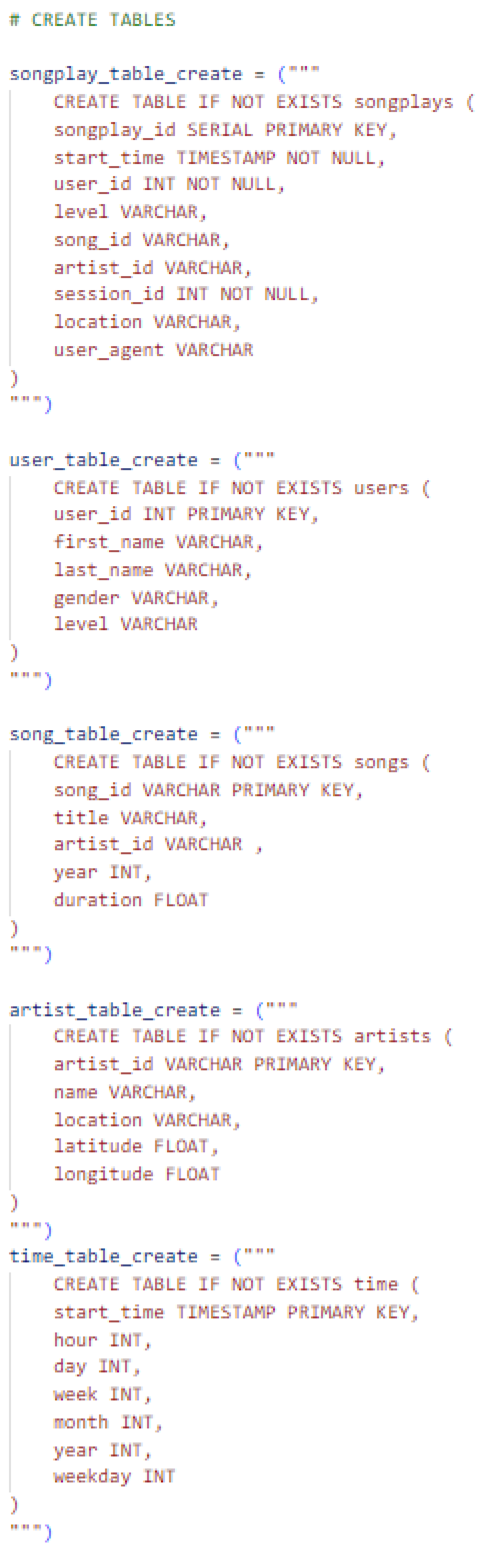
In order to create these tables, all we need to do is perform some transformation in the data which are already in the song_data and log_data directory.

4. **Drop tables if exist and Create tables queries**

Drop tables if tables exist and create tables in the database

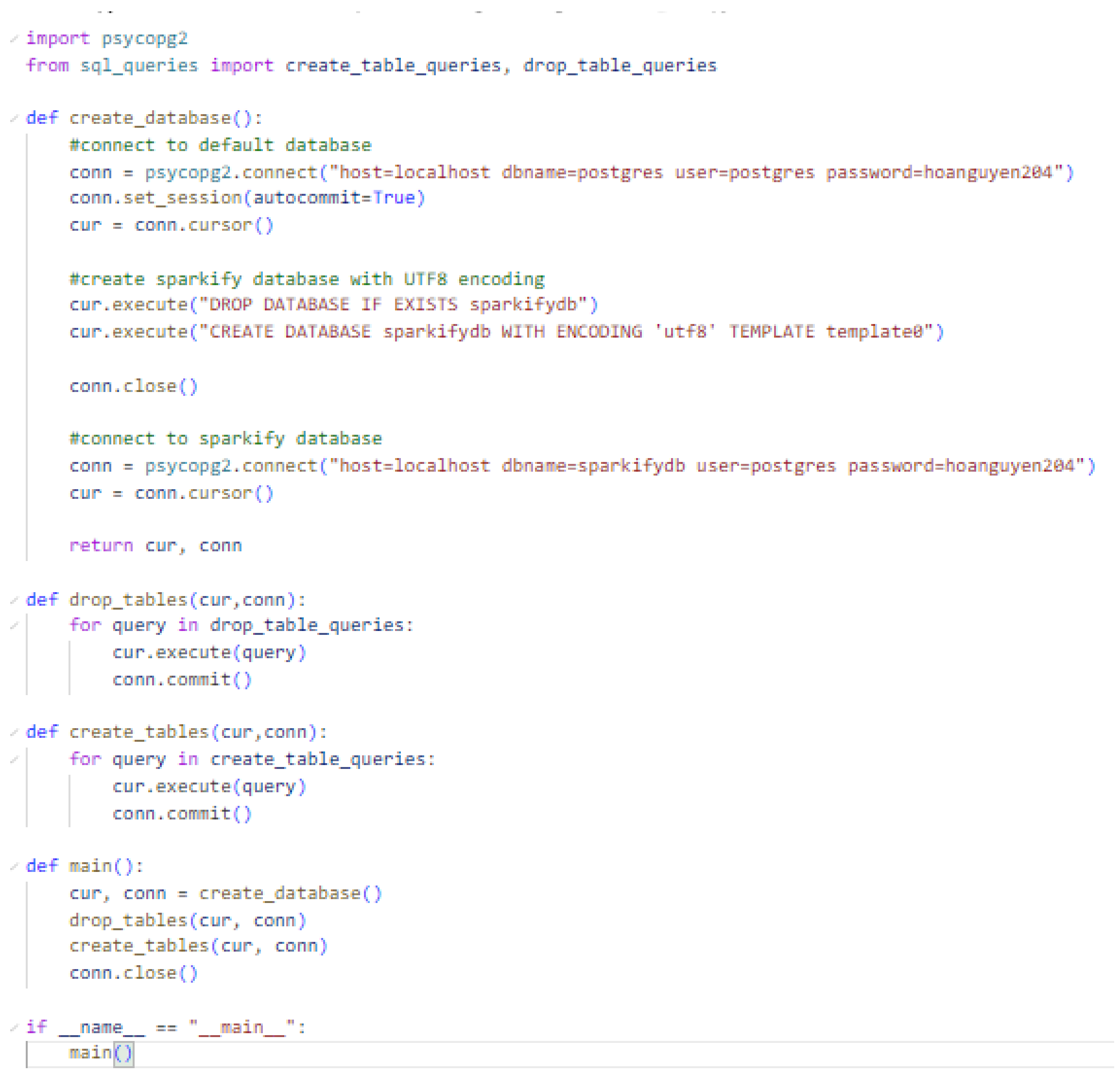


```
# DROP TABLES

songplay_table_drop = "DROP TABLE IF EXISTS songplays"
user_table_drop = "DROP TABLE IF EXISTS users"
song_table_drop = "DROP TABLE IF EXISTS songs"
artist_table_drop = "DROP TABLE IF EXISTS artists"
time_table_drop = "DROP TABLE IF EXISTS time"
```



```
# CREATE TABLES

songplay_table_create = ("""
    CREATE TABLE IF NOT EXISTS songplays (
    songplay_id SERIAL PRIMARY KEY,
    start_time TIMESTAMP NOT NULL,
    user_id INT NOT NULL,
    level VARCHAR,
    song_id VARCHAR,
    artist_id VARCHAR,
    session_id INT NOT NULL,
    location VARCHAR,
    user_agent VARCHAR
)
""")

user_table_create = ("""
    CREATE TABLE IF NOT EXISTS users (
    user_id INT PRIMARY KEY,
    first_name VARCHAR,
    last_name VARCHAR,
    gender VARCHAR,
    level VARCHAR
)
""")

song_table_create = ("""
    CREATE TABLE IF NOT EXISTS songs (
    song_id VARCHAR PRIMARY KEY,
    title VARCHAR,
    artist_id VARCHAR ,
    year INT,
    duration FLOAT
)
""")

artist_table_create = ("""
    CREATE TABLE IF NOT EXISTS artists (
    artist_id VARCHAR PRIMARY KEY,
    name VARCHAR,
    location VARCHAR,
    latitude FLOAT,
    longitude FLOAT
)
""")
time_table_create = ("""
    CREATE TABLE IF NOT EXISTS time (
    start_time TIMESTAMP PRIMARY KEY,
    hour INT,
    day INT,
    week INT,
    month INT,
    year INT,
    weekday INT
)
""")
```

5. **Connect to database and run Create database queries**
   In this file, we create a connection to postgre database and run drop and create tables queries in the front part.

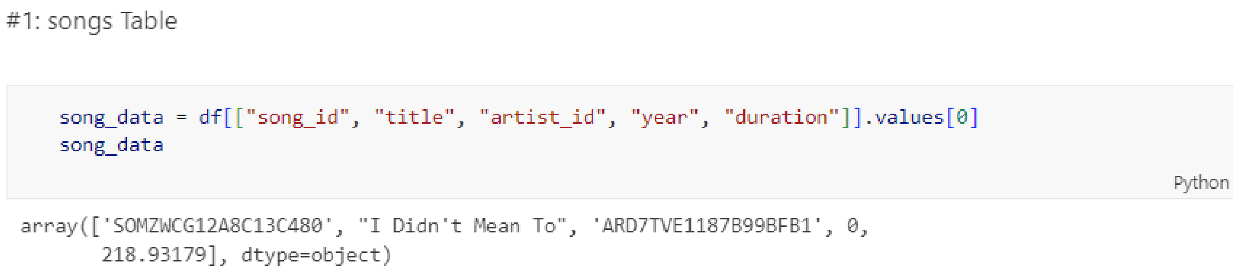


```
import psycopg2
from sql_queries import create_table_queries, drop_table_queries

def create_database():
    #connect to default database
    conn = psycopg2.connect("host=localhost dbname=postgres user=postgres password=hoanguyen204")
    conn.set_session(autocommit=True)
    cur = conn.cursor()

    #create sparkify database with UTF8 encoding
    cur.execute("DROP DATABASE IF EXISTS sparkifydb")
    cur.execute("CREATE DATABASE sparkifydb WITH ENCODING 'utf8' TEMPLATE template0")

    conn.close()

    #connect to sparkify database
    conn = psycopg2.connect("host=localhost dbname=sparkifydb user=postgres password=hoanguyen204")
    cur = conn.cursor()

    return cur, conn

def drop_tables(cur,conn):
    for query in drop_table_queries:
        cur.execute(query)
        conn.commit()

def create_tables(cur,conn):
    for query in create_table_queries:
        cur.execute(query)
        conn.commit()

def main():
    cur, conn = create_database()
    drop_tables(cur, conn)
    create_tables(cur, conn)
    conn.close()

if __name__ == "__main__":
    main()
```

6. **Process song data (song_data directory)**

   We will perform ETL on the files in *song_data* directory to create two dimensional tables: songs table and artists table

   For *songs table*, we'll extract data for *songs table* by using only the columns corresponding to the songs table suggested in the star schema above. Similarly, we'll select the appropriate columns for *artists table.*

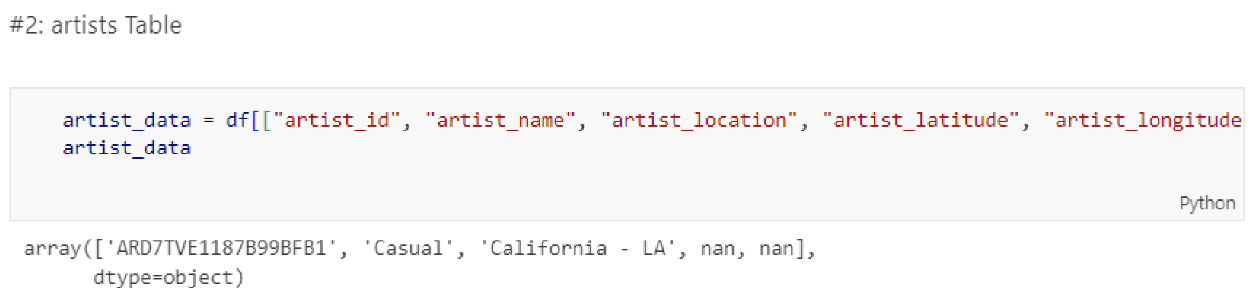


#1: songs Table

```
song_data = df[["song_id", "title", "artist_id", "year", "duration"]].values[0]
song_data
```

Python

```
array(['SOMZWCG12A8C13C480', "I Didn't Mean To", 'ARD7TVE1187B99BFB1', 0,
       218.93179], dtype=object)
```

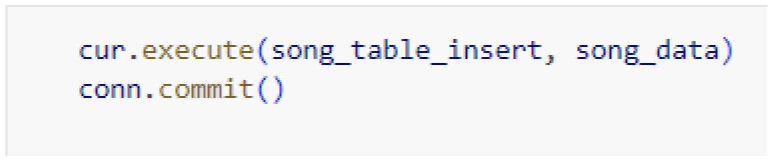

#2: artists Table

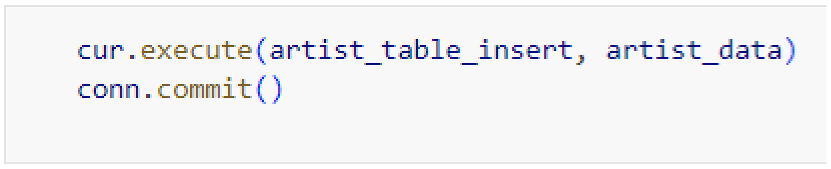
```
artist_data = df[["artist_id", "artist_name", "artist_location", "artist_latitude", "artist_longitude
artist_data
```

Python

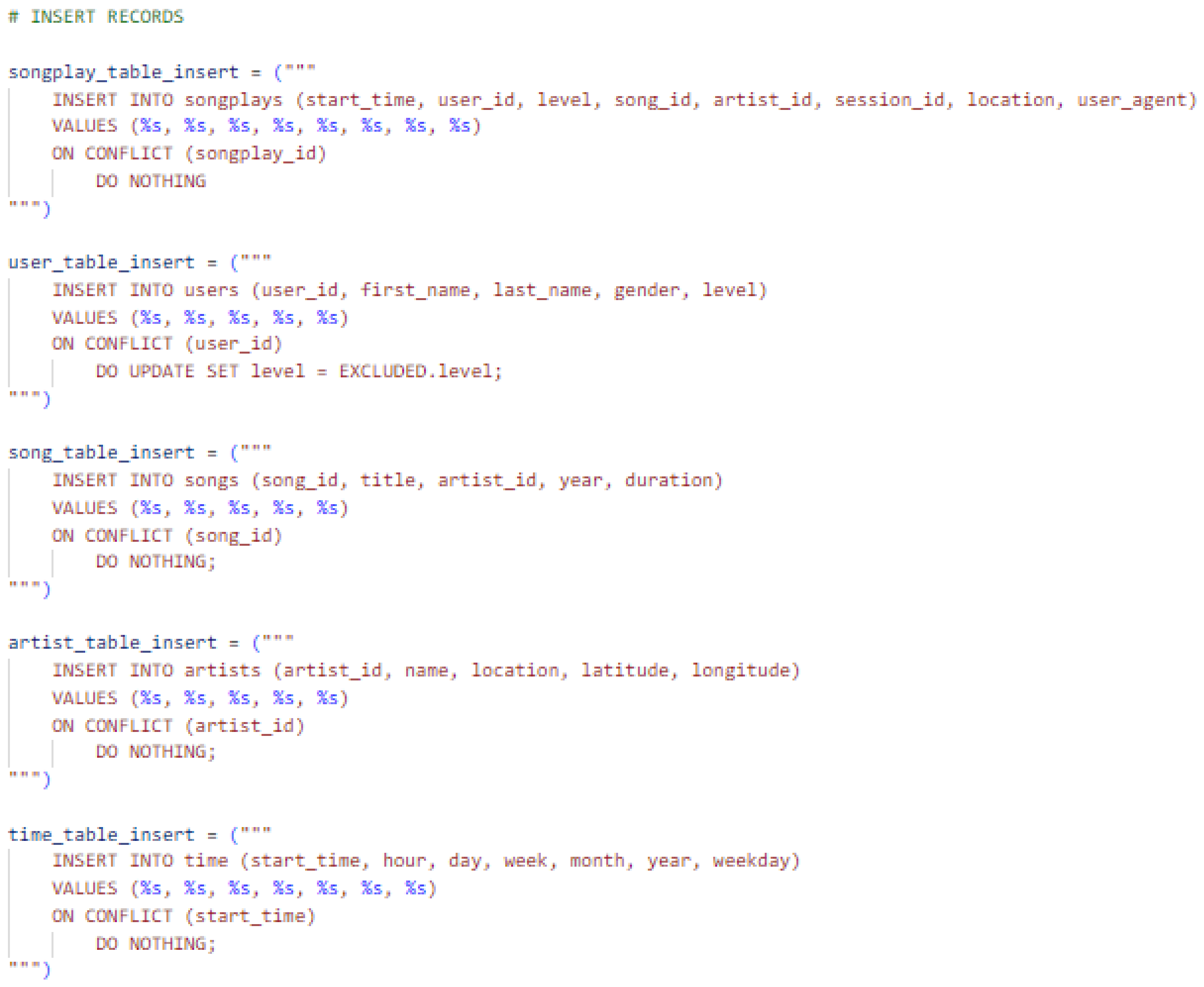
```
array(['ARD7TVE1187B99BFB1', 'Casual', 'California - LA', nan, nan],
      dtype=object)
```

Now insert the extract data into their respective tables.


```
cur.execute(song_table_insert, song_data)
conn.commit()
```


```
cur.execute(artist_table_insert, artist_data)
conn.commit()
```

Variables song_table_insert and artist_table_insert are SQL queries. These are given in *sql_queries.py* file.



```
# INSERT RECORDS

songplay_table_insert = ("""
    INSERT INTO songplays (start_time, user_id, level, song_id, artist_id, session_id, location, user_agent)
    VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
    ON CONFLICT (songplay_id)
        DO NOTHING
""")

user_table_insert = ("""
    INSERT INTO users (user_id, first_name, last_name, gender, level)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (user_id)
        DO UPDATE SET level = EXCLUDED.level;
""")

song_table_insert = ("""
    INSERT INTO songs (song_id, title, artist_id, year, duration)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (song_id)
        DO NOTHING;
""")

artist_table_insert = ("""
    INSERT INTO artists (artist_id, name, location, latitude, longitude)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (artist_id)
        DO NOTHING;
""")

time_table_insert = ("""
    INSERT INTO time (start_time, hour, day, week, month, year, weekday)
    VALUES (%s, %s, %s, %s, %s, %s, %s)
    ON CONFLICT (start_time)
        DO NOTHING;
""")
```

7. **Process log data (log_data directory)**
   We will perform ETL on the files in *log_data* directory to create the remaining two dimensional tables: `time` and `users`, as well as the `songplays` fact table.
   For `time` table we have `ts` column in log files. We will parse it as a time stamp and use python's datetime functions to create the remaining columns required for the table mentioned in the above schema.

```
t = pd.to_datetime(df['ts'], unit='ms')
t.head()
```

```
time_data = (t, t.dt.hour, t.dt.day, t.dt.isocalendar().week, t.dt.month, t.dt.year, t.dt.weekday)
column_labels = ('timestamp', 'hour', 'day', 'week of year', 'month',' year', 'weekday')
```

```
time_dict = dict(zip(column_labels, time_data))
time_df = pd.DataFrame.from_dict(time_dict)
time_df.head()
```

For songplays table, we will require information from songs table, artists table and the original log files. Since the log files do not have song_id and artist_id, we need to use songs table and artists table for that. The *song_select* query finds the song_id and artist_id based on the *title, artist_name,* and *duration* of a song. For the remaining columns, we can select them from the log files.

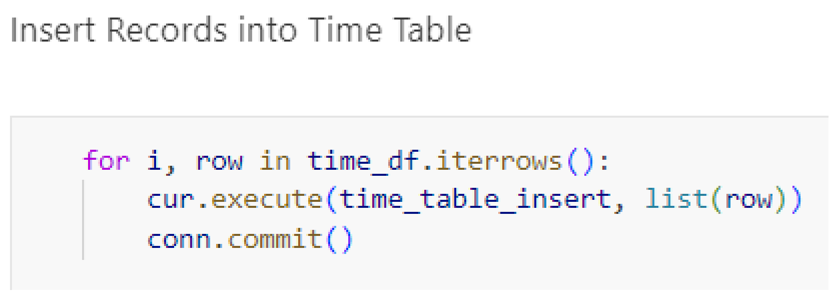
#4: users Table

```
user_df = df[['userId', 'firstName', 'lastName', 'gender', 'level']]
```

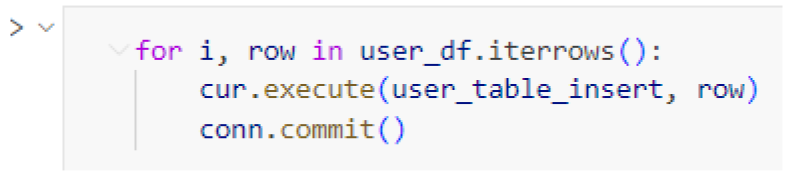
#5: songplays Table

```
for index, row in df.iterrows():

    #get songid and artistid from song and artist tables
    cur.execute(song_select, (row.song, row.artist, row.length))
    results = cur.fetchone()

    if results:
        songid, artistid = results
    else:
        songid, artistid = None, None
```

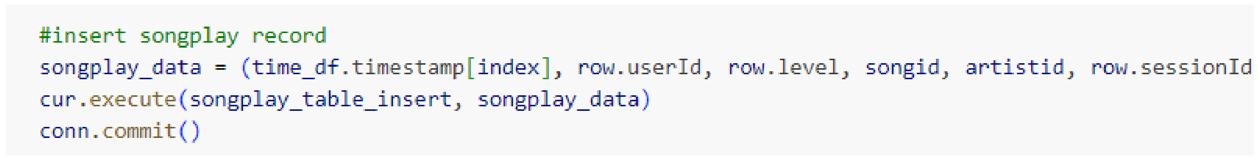
Now insert the data into their respective tables.


Insert Records into Time Table

```
for i, row in time_df.iterrows():
    cur.execute(time_table_insert, list(row))
    conn.commit()
```



```
for i, row in user_df.iterrows():
    cur.execute(user_table_insert, row)
    conn.commit()
```



```
#insert songplay record
songplay_data = (time_df.timestamp[index], row.userId, row.level, songid, artistid, row.sessionId
cur.execute(songplay_table_insert, songplay_data)
conn.commit()
```

## 8. Conclusion

We created a Postgres database with the facts and dimension table for song_play analysis. We populated it with the entries from songs and events directory. Now our data is useful for some basic aggregation and analytics.