Task One:

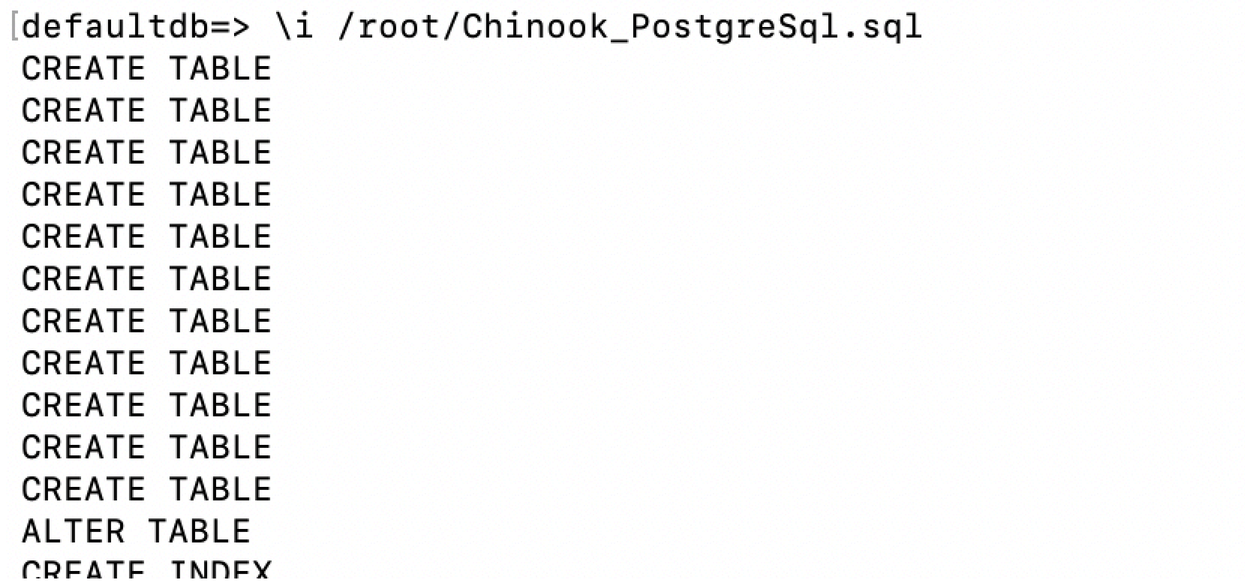
> Performed the first task using Hasura cloud and used my DigitalOcean postgres database as a data source.

Steps taken to create the environment :

Connection to my database was pretty easy as I used the database URL directly to connect to my database from Hasura. I navigated to Hasura Cloud and followed the below steps:
> Create a new project
> Launch Console
> Data tab
and configured the name of DB, type of DB and selected database URL to establish the connection.

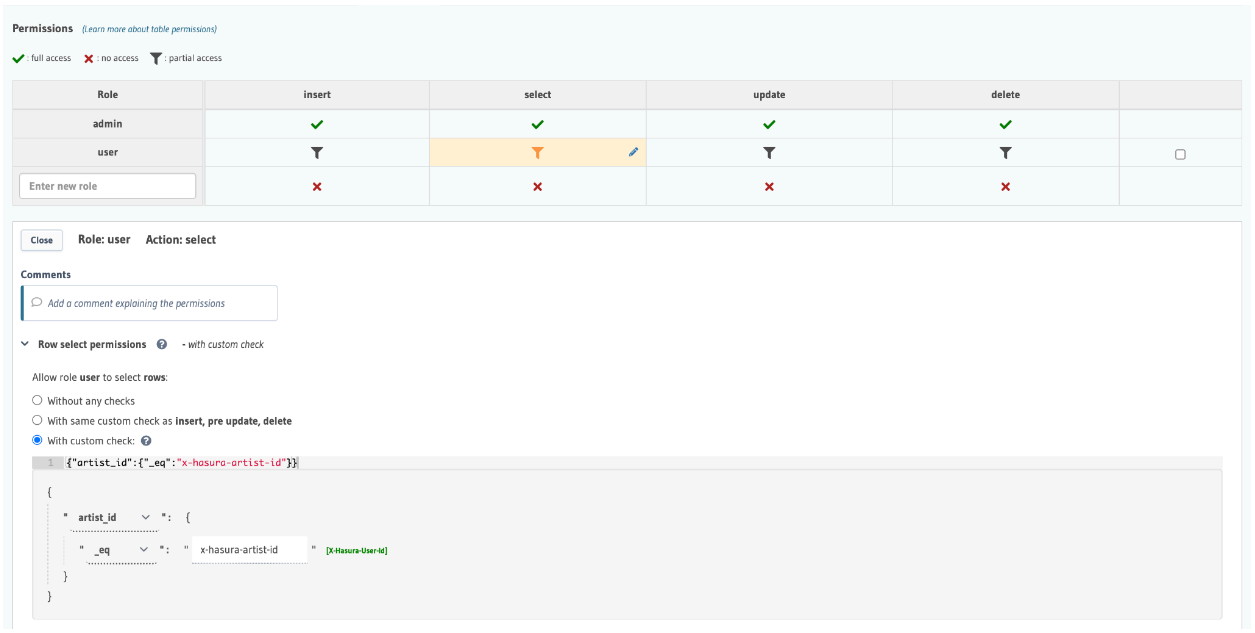
Further, generated the connection string for postgres using doctl api.
Command Used : doctl databases connection <database-cluster-id>

After the DB was connected to Hasura, I logged in to DB using psql client and used the chinook data set for postgres. By downloading the dump from the github that was shared in the worksheet provided, I then migrated the data to my postgres database.



```
[defaultdb=> \i /root/Chinook_PostgreSql.sql
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 CREATE TABLE
 ALTER TABLE
 CREATE INDEX
```

However, I faced a challenge with my tables not getting listed on Hasura cloud. I initially thought that was due to an access issue to the chinook database after table creation. To resolve this, I implemented a workaround by not creating a new database. Albeit, creating the tables directly into defaultDB which allowed me to list the tables in Hasura. In order for the user to fetch the data without using Admin secret, I created a role named user and assigned the adequate permissions on the album table. To ensure that users are only able to access the albums owned by them, I further created the below check for x-hasura-artist-id.



Permissions *(Learn more about table permissions)*

✓ : full access ✗ : no access ▼ : partial access

Role	insert	select	update	delete	
admin	✓	✓	✓	✓	
user	▼	▼	▼	▼	☐
Enter new role	✗	✗	✗	✗	

Close **Role: user** **Action: select**

Comments

Add a comment explaining the permissions

Row select permissions *- with custom check*

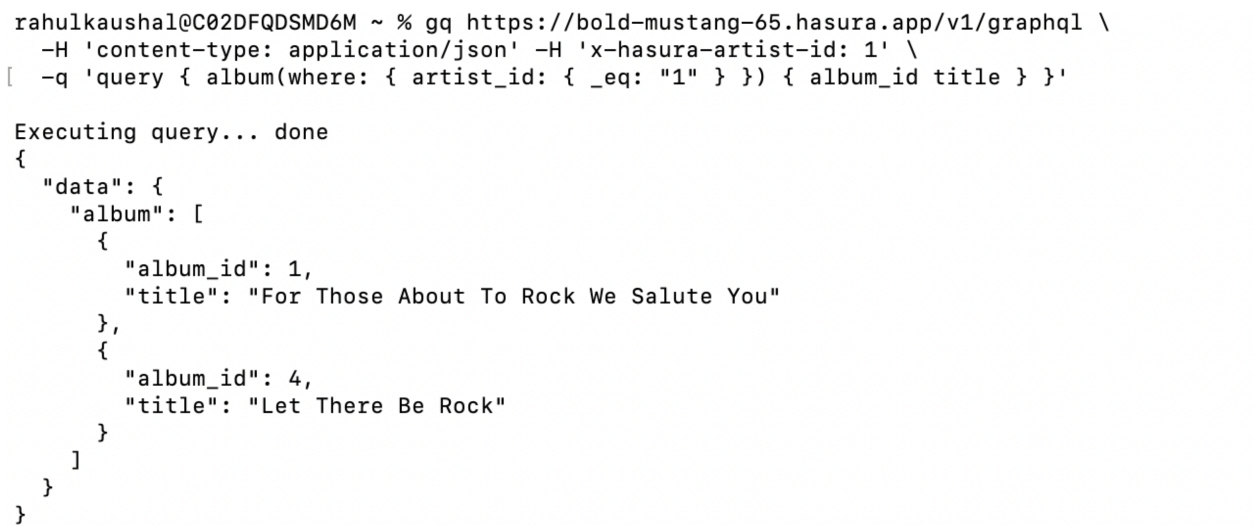
Allow role **user** to select **rows**:

- ○ Without any checks
- ○ With same custom check as **insert, pre update, delete**
- ● With custom check:

```
{"artist_id":{"_eq":"x-hasura-artist-id"}}
```

```
{
  " artist_id ": {
    " _eq ": " x-hasura-artist-id " [X-Hasura-User-Id]
  }
}
```

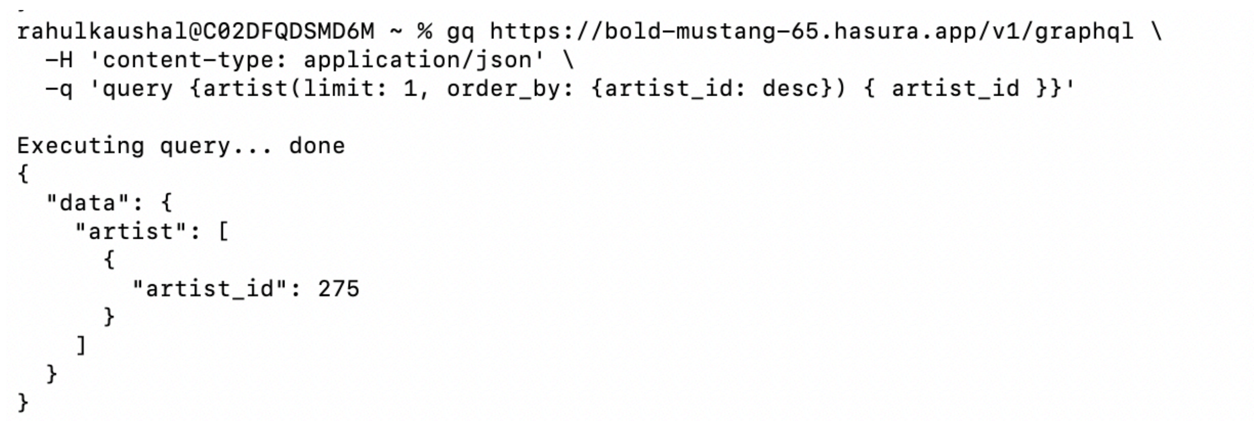
and it worked



```
rahulkaushal@C02DFQDSMD6M ~ % gq https://bold-mustang-65.hasura.app/v1/graphql \
  -H 'content-type: application/json' -H 'x-hasura-artist-id: 1' \
[  -q 'query { album(where: { artist_id: { _eq: "1" } }) { album_id title } }'

Executing query... done
{
  "data": {
    "album": [
      {
        "album_id": 1,
        "title": "For Those About To Rock We Salute You"
      },
      {
        "album_id": 4,
        "title": "Let There Be Rock"
      }
    ]
  }
}
```

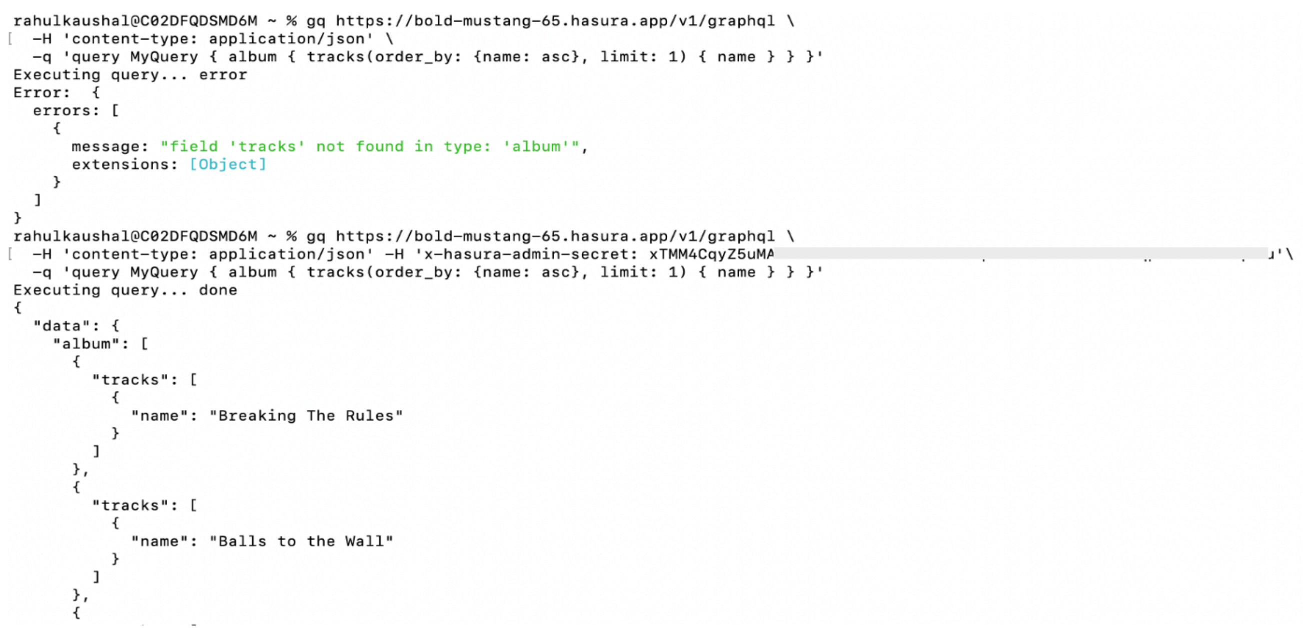
Deliverables

Question 1 How many artists are in the database?
Answer: To determine the total number of artists in the database, I utilized the artist_ID field. Since GraphQL lacks a built-in method for such aggregation, I devised a workaround. Knowing that the IDs were arranged in ascending order, I employed the following strategy to ascertain the total artists: I queried the database, limiting the result to one entry and sorting it in descending order. This approach ensured that the last artist ID retrieved represented the total count of artists in the table. Consequently, the total number of artists in the database was found to be 275.



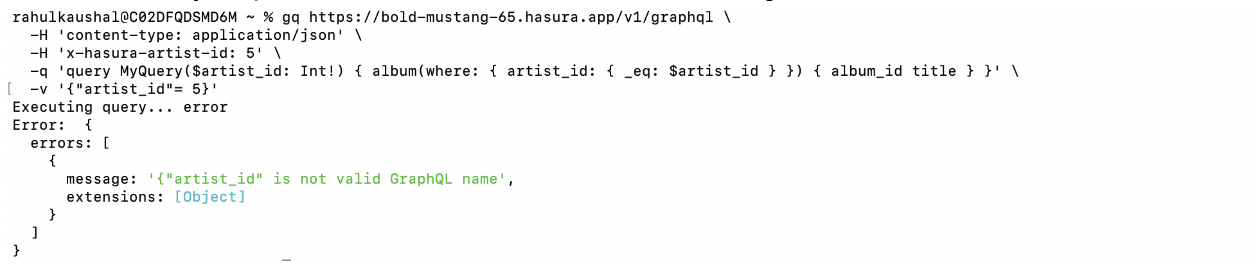
```
rahulkaushal@C02DFQDSMD6M ~ % gq https://bold-mustang-65.hasura.app/v1/graphql \
  -H 'content-type: application/json' \
  -q 'query {artist(limit: 1, order_by: {artist_id: desc}) { artist_id }}'

Executing query... done
{
  "data": {
    "artist": [
      {
        "artist_id": 275
      }
    ]
  }
}
```

Question 2: List the first track of every album by every artist in ascending order.
Answer: First of all I created a relationship by navigating to Data within the tables artist, album and track. Later I used the below query to fetch the tracks and their names in ascending order. However, it seems due to some permission issue. I am only able to fetch this data in case I use Hasura admin. Might be due to some permission issue I have created for the tables. Unfortunately I was not able to figure it out due to lack of time.



```
rahulkaushal@C02DFQDSMD6M ~ % gq https://bold-mustang-65.hasura.app/v1/graphql \
[  -H 'content-type: application/json' \
  -q 'query MyQuery { album { tracks(order_by: {name: asc}, limit: 1) { name } } }'
Executing query... error
Error:  {
  errors: [
    {
      message: "field 'tracks' not found in type: 'album'",
      extensions: [Object]
    }
  ]
}
rahulkaushal@C02DFQDSMD6M ~ % gq https://bold-mustang-65.hasura.app/v1/graphql \
[  -H 'content-type: application/json' -H 'x-hasura-admin-secret: xTMM4CqyZ5uMA[redacted]'\
  -q 'query MyQuery { album { tracks(order_by: {name: asc}, limit: 1) { name } } }'
Executing query... done
{
  "data": {
    "album": [
      {
        "tracks": [
          {
            "name": "Breaking The Rules"
          }
        ]
      },
      {
        "tracks": [
          {
            "name": "Balls to the Wall"
          }
        ]
      },
      {
```

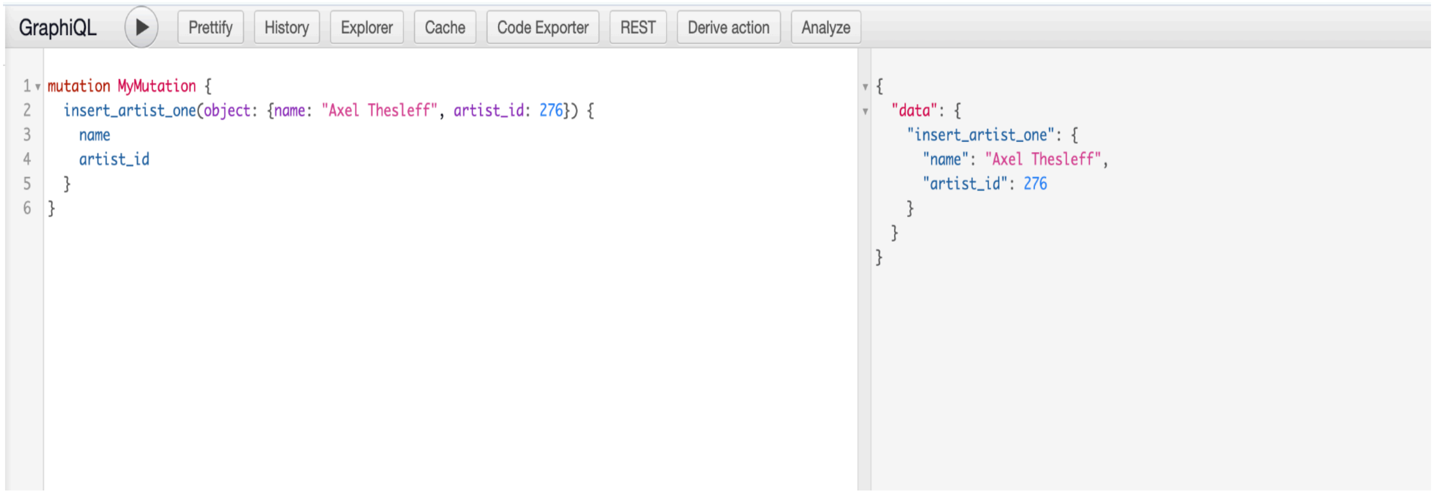
Question 3: Get all albums for artist id = 5 without specifying a where clause.
Answer: For this question I understood I have to use variables to pass the value for the artist_id in case I don't want to use the where clause. But unfortunately, I was not able to find the way to parse a variable below is the error I get



```
rahulkaushal@C02DFQDSMD6M ~ % gq https://bold-mustang-65.hasura.app/v1/graphql \
  -H 'content-type: application/json' \
  -H 'x-hasura-artist-id: 5' \
  -q 'query MyQuery($artist_id: Int!) { album(where: { artist_id: { _eq: $artist_id } }) { album_id title } }' \
[  -v '{"artist_id"= 5}'
Executing query... error
Error:  {
  errors: [
    {
      message: '{"artist_id" is not valid GraphQL name',
      extensions: [Object]
    }
  ]
}
```

Question 4: Using a GraphQL mutation, add your favorite artist and one of their albums that isn't in the dataset.
Answer: Added a new artist Axel Thesleff to the artist table and the album Two World to the album below are the GraphQL queries I used

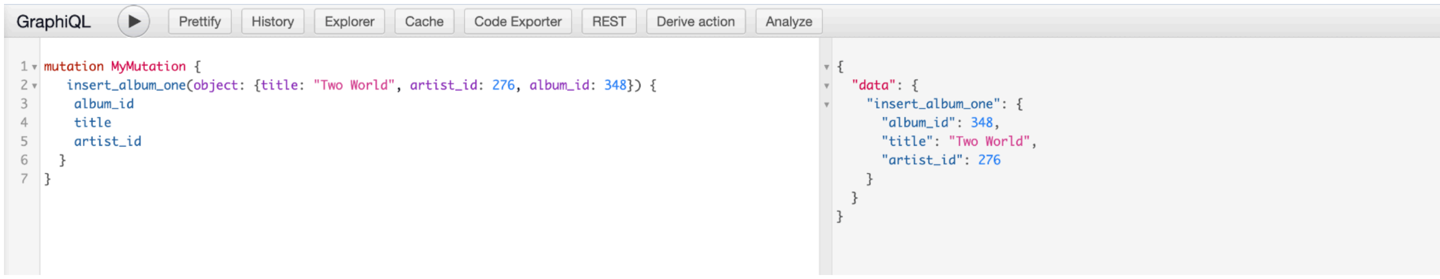
Mutation to Artist:


GraphiQL Prettify History Explorer Cache Code Exporter REST Derive action Analyze

```
mutation MyMutation {
  insert_artist_one(object: {name: "Axel Thesleff", artist_id: 276}) {
    name
    artist_id
  }
}
```

```
{
  "data": {
    "insert_artist_one": {
      "name": "Axel Thesleff",
      "artist_id": 276
    }
  }
}
```

Mutation to Album:

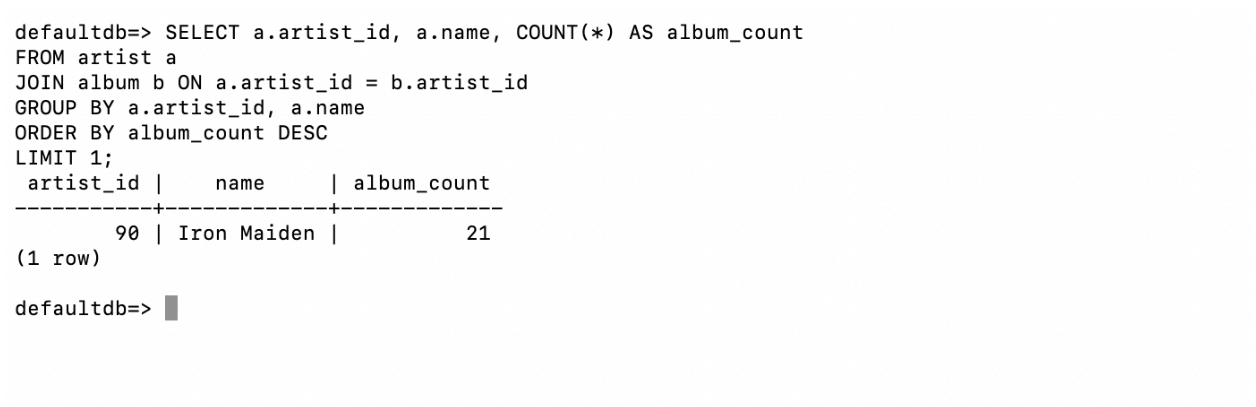

GraphiQL Prettify History Explorer Cache Code Exporter REST Derive action Analyze

```
mutation MyMutation {
  insert_album_one(object: {title: "Two World", artist_id: 276, album_id: 348}) {
    album_id
    title
    artist_id
  }
}
```

```
{
  "data": {
    "insert_album_one": {
      "album_id": 348,
      "title": "Two World",
      "artist_id": 276
    }
  }
}
```

Question 5: How did you identify which ID to include in the mutation?
Answer: I required 2 unique ID's for album_ID and artist_ID. As both the IDs were in ascending order I picked out the mac by querying the ID from descending and limiting it to one. It gave me the last available ID.I incremented this by 1 and was able to get the unique ID for adding my album ID and the artist ID.

Postgres question 1:Return the artist with the most number of albums
Answer: Iron maiden is the artist with the most albums. Below is query and the output from the psql client:

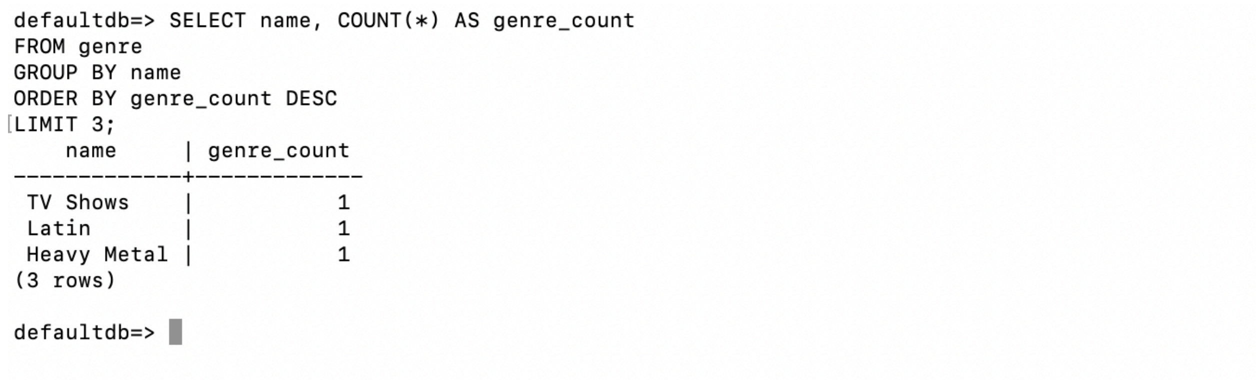


```
defaultdb=> SELECT a.artist_id, a.name, COUNT(*) AS album_count
FROM artist a
JOIN album b ON a.artist_id = b.artist_id
GROUP BY a.artist_id, a.name
ORDER BY album_count DESC
LIMIT 1;
 artist_id |    name     | album_count
-----------+-------------+-------------
        90 | Iron Maiden |          21
(1 row)

defaultdb=> 
```

Initially, I retrieve the artist_id and name from the artist table, also counting the number of albums for each artist using COUNT(*) as album_count. Subsequently, I join the artist and album tables based on the artist_id column. The result is grouped by artist_id and name, then sorted in descending order by the album count. Finally, to ascertain the artist with the highest number of albums, the result is restricted to a single row.

Postgres Question 2: Return the top three genres found in the dataset in descending order

Answer: Below is the output from the psql client

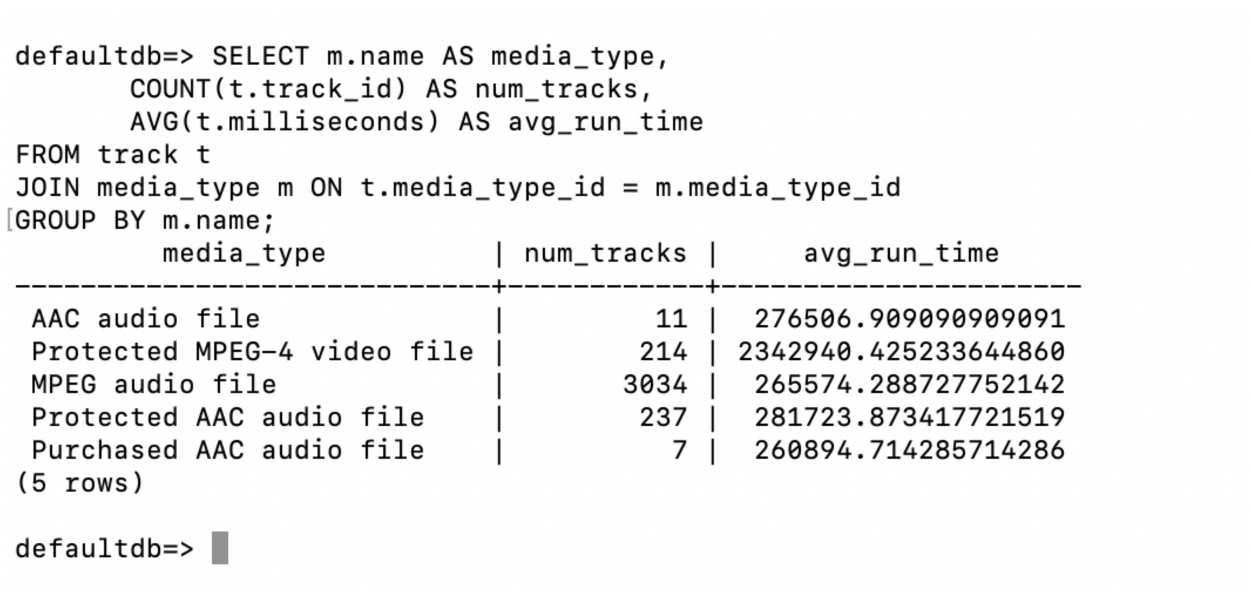


```
defaultdb=> SELECT name, COUNT(*) AS genre_count
FROM genre
GROUP BY name
ORDER BY genre_count DESC
LIMIT 3;
    name     | genre_count
-------------+-------------
 TV Shows    |           1
 Latin       |           1
 Heavy Metal |           1
(3 rows)

defaultdb=> 
```

I start by retrieving unique genre names from the genre table. Then, calculate the number of occurrences for each genre using COUNT(*) and label it as genre_count. These counts are grouped by genre name. The result is sorted in descending order based on the count of occurrences. Lastly, we limit the result to include only the top three rows.

Postgres Question 3: Return the number of tracks and average run time for each media type

Answer: Below is the output for average run time of all media type



```
defaultdb=> SELECT m.name AS media_type,
       COUNT(t.track_id) AS num_tracks,
       AVG(t.milliseconds) AS avg_run_time
FROM track t
JOIN media_type m ON t.media_type_id = m.media_type_id
GROUP BY m.name;
         media_type          | num_tracks |     avg_run_time
-----------------------------+------------+-----------------------
 AAC audio file              |         11 |  276506.909090909091
 Protected MPEG-4 video file |        214 | 2342940.425233644860
 MPEG audio file             |       3034 |  265574.288727752142
 Protected AAC audio file    |        237 |  281723.873417721519
 Purchased AAC audio file    |          7 |  260894.714285714286
(5 rows)

defaultdb=> 
```

First, I start by selecting the media type name from the media_type table. Next, used the count function to tracks and compute the average run time associated with each media type. The track and media_type tables are connected through their media_type_id column via a join operation and the resulting data set is grouped based on the media type name.