Faculty of Sciences of the University of Lisbon

Database Technologies - Report 1

PostgreSQL vs MongoDB

Group 5 - António Rebelo · Duarte Balata · Filipa Serrano · Marco Mercier

taught by
Professor Cátia Pesquita

August 11, 2021

1 Introduction

The first step of the project was to analyse the type and content of the data to be worked with. The project data consisted of csv files with information regarding bands, their released albums, genres and former or active artists, extracted from dbpedia [1]. A light data cleaning process needed to be done in order to make the data cohese by removing undesirable words and characters, duplicated or broken entries and to change the date format into the SQL standard.

2 Implementation

The database was implemented with a relational model and with a NoSQL model in order to understand the main differences between them (in terms of modeling the data, querying, transactions and overall performance), and what would be the most appropriate for the provided data.

2.1 Relational Database

Relational databases are still the most widely used model, especially if the data has a lot of relations and a defined structure, or there is a need to maintain strict data integrity [6]. The reason for that is that SQL databases present what's known as ACID properties, meaning they are capable of ensuring the Atomicity, Consistency, Isolation and Durability of the data. Using PostgreSQL database management system (DBMS) [5] for the implementation of this work's relational database model, it was necessary to have a good understanding of the provided data structure. With that in mind, an Entity-Relationship Model was built.

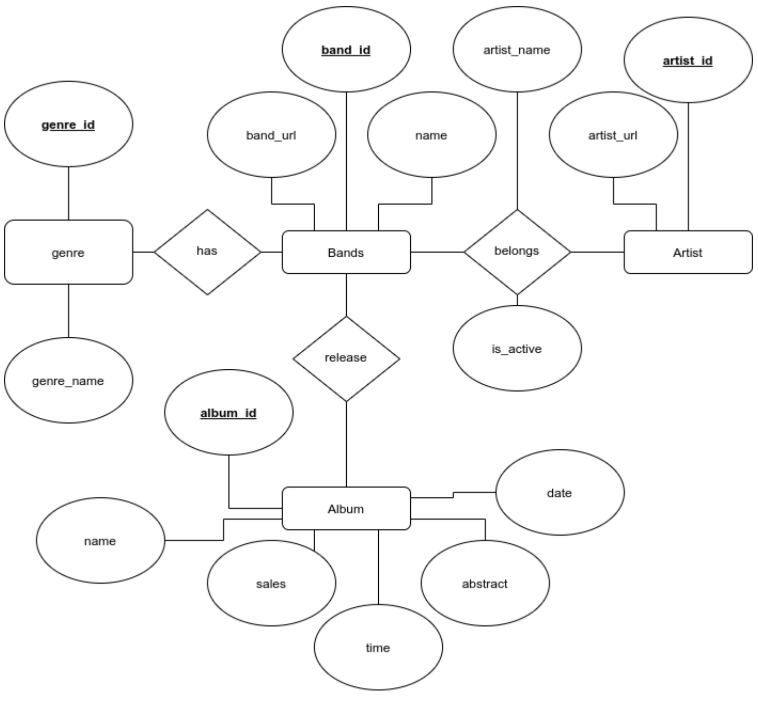


Figure 1: Entity-Relationship Model for the SQL database

For this type of model, a basic step is to decide whether the objects should be considered entities, attributes, or even relationship attributes in order to better describe the domain. While some of the objects were easy to classify as entities and its respective attributes, the object Genres led to some questions, as it was grouped with the albums in the original data, but each band can have a lot of different genres. It was decided that it would be a better option to consider it an entity itself that could establish a relationship set with the bands. Two other options were considered, but they both had some problems. The first alternative would be to have an array of genres associated with each band, but then the database wouldn't take advantage of SQL relational capabilities as the array would hide the structure. The other option was having a column for each genre, but since most bands only have a few genres, there would be a lot of null values and that would be a waste of memory. Besides, handling sparse data is not SQL's forte.

On the same note, it was decided that the status of former or active artists in a band would be represented by a boolean is_active, making it possible to have the same artist belonging to different bands with different statuses instead of having different entities to describe former and active band members. Regarding the artists' names, they were considered an attribute of the relationship between the artist and the band, instead of being an attribute of the artist himself. This decision was based on the fact that the artists can have different names depending on the band they are in.

It was decided that an _id attribute for each of the entities would be created for several reasons. First, it improves the performance when querying because you can easily see that comparing ids when joining, instead of the URLs (which are also unique), for example, is a lot faster since the latter is a relatively large string. Then, it makes setting up foreign key constraints a lot easier and finally, it makes querying more intuitive and the result is more readable. The IDs were set using PostgreSQL's type "serial", that guarantees the generation of a unique ID for each different entry.

Another important step is to connect the entities with relationships between them and define what type of relationships they are. The relationship between bands and artists is many-to-many, as each band can have multiple artists and each artist can belong (or have belonged in the past) to several bands. The relationship between genres and bands is also many-to-many, as each band can have multiple genres, and the same genre can relate to several different bands. The relationship between the bands and the albums could, at a first analysis, seems to be a one-to-many relationship, as one band produces many albums, and albums are usually produced by one band only. However, some albums, including some present in our dataset, are produced by more than one band, making this a many-to-may relationship.

After the general organization was established, in order to start the foundations for the relational database, it was necessary to translate it to a relational model as a collection of tables. The relation schema is represented in figure 2. The goal was to create relations that would be efficient in terms of storage, but at the same time allow an efficient querying of the data. For each table, it was also needed to determine all types of constraints, including unique, primary and foreign keys.

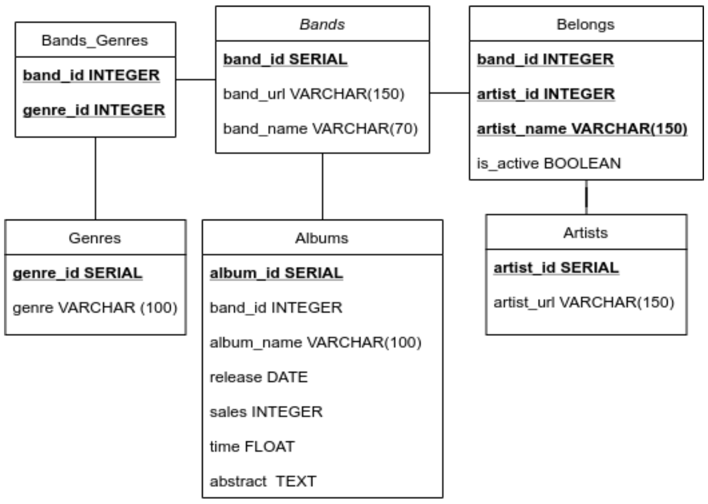


Figure 2: Relation schema for Relational Database

As a side note it should be observed that originally the data would violate the primary key of the table Belongs in the SQL database (see Fig. 2), because there were duplicate entries (in the PK fields) of artists that had left a certain band and then returned back to it. All fields except "is_active" were the same for these cases. Since there were only a handful of examples and it was considered that the information contained in that duplication was not very relevant, in order to maintain the integrity of the data, those entries with "is_active" = False would be erased. That specific information was thrown away but a primary key that better translates what constitutes the uniqueness of the artist is gained.

It is common to create a table that establishes the relationship between two entities which have a many-to-many relationship. That was done for most of the relationships in the database. However, in the case of the relationship between the albums and bands, because of the small proportion of albums which were produced by more than one band, it was decided that it was preferable to simplify the model by avoiding that table, with the tradeoff of having some repetition of album entries. Still regarding the types, most strings were set as "varchar" with a margin of characters that would not only allow the insertion of all the data at the time, but also with some extra room for potentially larger future entries; the albums abstracts were set as "text", since they are large strings that would be hard to define otherwise; and the time of each album as float representing the length of the albums in minutes. The other types were fairly intuitive to establish. To prepare the data to be uploaded to the database, it was necessary to write a script that created csv files that contained the data corresponding to each table by reorganizing and processing the original csv files provided. A script using python was written in order to do so. The CREATE TABLE and COPY statements to insert the data into the tables, were written as in the appendix or in the create_tables.sql file in the scripts folder.

2.2 NoSQL Database

MongoDB is a document-oriented non-relational DBMS [2]. MongoDB's advantages consist of its capability to scale horizontally and support for large quantities of read and write transactions, which make it an excellent option for people working with large quantities of internet data. Apart from that, there are some other advantages to using document-oriented databases. For example, when there's a need to frequently access clusters of data, it's advantageous that the information is stored in a single location, instead of spread out across multiple tables in a relational database, from where it would have to be fetched using 'join' statements, that are not only syntactically complex, but also computationally expensive. The use of NoSQL database is also to be considered when data parameters are unpredictable or subject to frequent change, since in relational type databases, the data structure must be predefined in a schema, whereas non-relational databases give the developers the freedom to add new parameters at will, making it possible for application developers to adjust quickly in the present fast-paced technology landscape.

Even though NoSQL document-oriented databases don't follow a rigid schema like observed on the relational model, the data is still somewhat structured [**nosql**]. For that reason, in order to create a Mongo database, it is necessary to define an aggregation schema that dictates in which document each parameter must be stored. Given that, it was decided that a single aggregate (per band) would work best for the data presented in this work and the type of queries that would be made, which included interactions between all of the available data fields. That choice was based on the known fact that aggregate oriented databases typically handle intra-aggregate relations better than inter-aggregate ones when querying. The final aggregation scheme is depicted on figure 3.

MongoDB's document data model can be represented in a JSON format file, but in order to optimize for speed, space and flexibility the data is internally stored in 'binary JSON' format (BSON). Since documents can be represented by a JSON file and Mongo is prepared to handle this type of file, data insertion was very straight forward using Python, since Python's dictionaries' data organization is very close to that of a JSON file. For that reason, with a single short Python script (add_mongo.py) [3], it was possible to parse all the raw data from the CSV files into Python dictionaries and directly import them into MongoDB Atlas, an on-demand fully managed database service for online storage that runs on AWS, Microsoft Azure, and Google Cloud Platform.

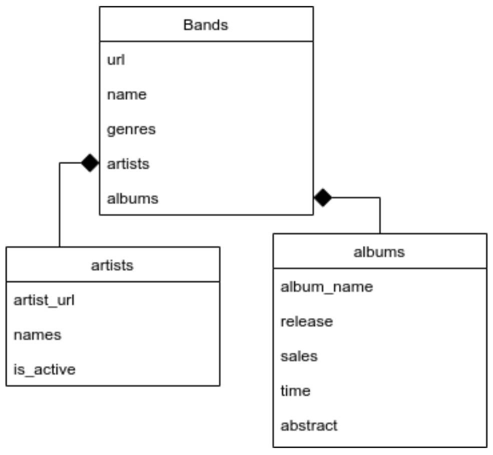


Figure 3: Aggregation scheme for MongoDB

3 Query Implementation

The chosen queries to compare the relational database and the NoSQL database require relationships and selection of data between several elements of the database, so they are capable of highlighting the main differences between both types of databases.

Query 1: Which artists belong/have belonged to the "Indie rock" band which has released the longest albums on average?

For the SQL implementation, the query was the following:

```
CREATE VIEW Bands_Average AS SELECT A.band_id AS band_id, AVG(A.time) AS avg_lenght
                         FROM Albums A
                         GROUP BY A.band_id
                             HAVING A.band_id IN (SELECT BA.band_id
                                                  FROM Bands BA, Bands_Genres BG, Genres G
                                                  WHERE BA.band_id = BG.band_id
                                                  AND BG.genre_id = G.genre_id
                                                  AND G.genre LIKE 'Indie rock');
SELECT DISTINCT BE.artist_name
FROM Belongs BE , Bands_Average
WHERE BE.band_id = Bands_Average.band_id AND
Bands_Average.avg_lenght = (SELECT MAX(Bands_Average.avg_lenght)
FROM Bands_Average);
```

Starting by the inner nest in the Bands_Average view, the band_id's that have the genre "Indie rock" are selected, then the average album length is calculated for each band_id. So Bands_Average has the columns Band_id|Average_album_length. Then the band that has the maximum average album length is selected, and finally the artists from that band are selected using the band_id.

For the MongoDB Implementation, this was the query:

```
db.Bands.aggregate([
        {"$match": {"genres":"Indie rock"}},
        {"$project": {
                "_id":0,
                "AvgTime": {"$avg":"$albums.time"},
                "name":1 ,
                "artists.names":1 }},
        {"$sort":{"AvgTime":-1 }},
        {"$limit":1} ]).pretty()
```

Using mongoDB's aggregate pipeline [4], the query starts by matching the genre "Indie rock" and, for the matching results, it projects the average album time, band name and artists names. The last step was to sort the document by descending average album time and limit the output to the first result.

Query 2:Which musical genre has the biggest number of albums released on each decade?

For the SQL implementation, the query was the following:

```
SELECT Ranking.decade AS "Decade", Ranking.genre AS "Genre", Ranking.album_nr AS "Album_nr"
FROM (SELECT Decade_Genre.decade AS decade, Decade_Genre.genre AS genre,
        RANK() OVER (PARTITION BY Decade_Genre.decade ORDER BY Decade_Genre.album_nr DESC) album_rank,
        Decade_Genre.album_nr as album_nr
        FROM (SELECT (EXTRACT (DECADE FROM A.release)) AS decade, G.genre AS genre,
        COUNT(A.album_id) AS album_nr
                FROM Albums A, Bands BA, Bands_Genres BG, Genres G
                WHERE A.band_id = BA.band_id
                AND BA.band_id = BG.band_id
                AND BG.genre_id = G.genre_id
                GROUP BY (EXTRACT (DECADE FROM A.release)), G.genre
                ORDER BY (EXTRACT (DECADE FROM A.release)) ) AS Decade_Genre)  AS Ranking
WHERE Ranking.album_rank = 1;
```

Again, starting from the inner layer of the query, firstly the decade for each album is obtained with the EXTRACT function and then an aggregation operation is performed to get the table Decade_Genre with columns

(Decade|Genre|Album_nr). Then by partitioning the data by decade, a ranking for each genre is obtained based on the number of released albums. This is put into the Ranking table (Decade|Genre|Album_rank|Album_nr). Finally, the decades, genres and number of albums where the rank is 1 are selected.

For the MongoDB Implementation, this was the query:

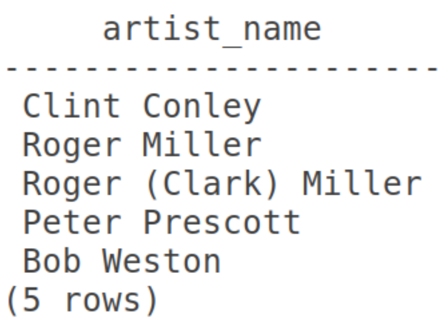
```
db.Bands.aggregate([
        {"$unwind":"$genres"},
        {"$unwind":"$albums"},
        {"$addFields":{"Year":{"$year": "$albums.release"}}},
        {"$bucket":{groupBy: "$Year",
boundaries:[1920,1930,1940,1950,1960,1970,1980,1990,2000,2010,2020],
                default:"Other",
                output: {"genres":{"$push":{"g":"$genres"}}}}},
        {"$unwind":"$genres"},
        {"$unwind":"$genres.g"},
        {"$group":{_id:{"Decade":"$_id", "Genre":"$genres.g"},count:{"$sum":1}}},
        {"$sort":{"_id.Decade":1,"count":-1}} ,
        {"$group":{_id:"$_id.Decade",
                "Top genre":{"$first":"$_id.Genre"},
                "Albums":{"$first":"$count"}}} ,
        {"$sort":{"_id":1}}]).pretty()
```

MongoDB's aggregate pipeline starts by unwinding the genres and albums arrays, turning each genre and album a different document. After that the year is extracted from the album release date and a bucket for each decade is created, pushing the genres of each decade into a single "genres" array. The idea then was to count how many genres of each genre are in each decade and to do that, the genres array was unwinded. The count for Decade/Genre is given by grouping the output by Decade and Genre. Sorting this group makes it possible to get the entries Decade/Genre by the total number of albums and grouping again just by Decade makes it possible to print just the top genre and the number of albums.

It should be noted, that it was discovered that in the 50's there's a tie for two genres. Even though thorough research was made to be able to display both results, like SQL does, it was not possible to solve this issue in mongoDB.

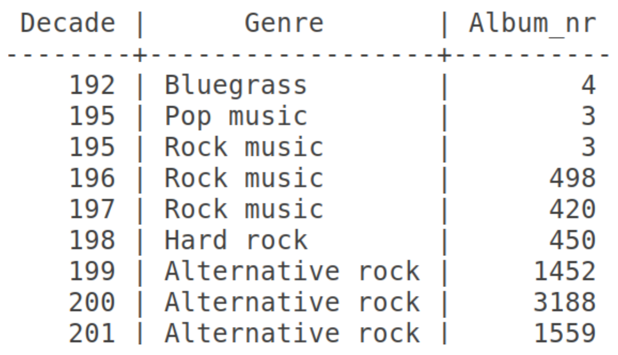
4 Results

In the following section, the results of the queries are presented.



```
     artist_name
----------------------
 Clint Conley
 Roger Miller
 Roger (Clark) Miller
 Peter Prescott
 Bob Weston
(5 rows)
```

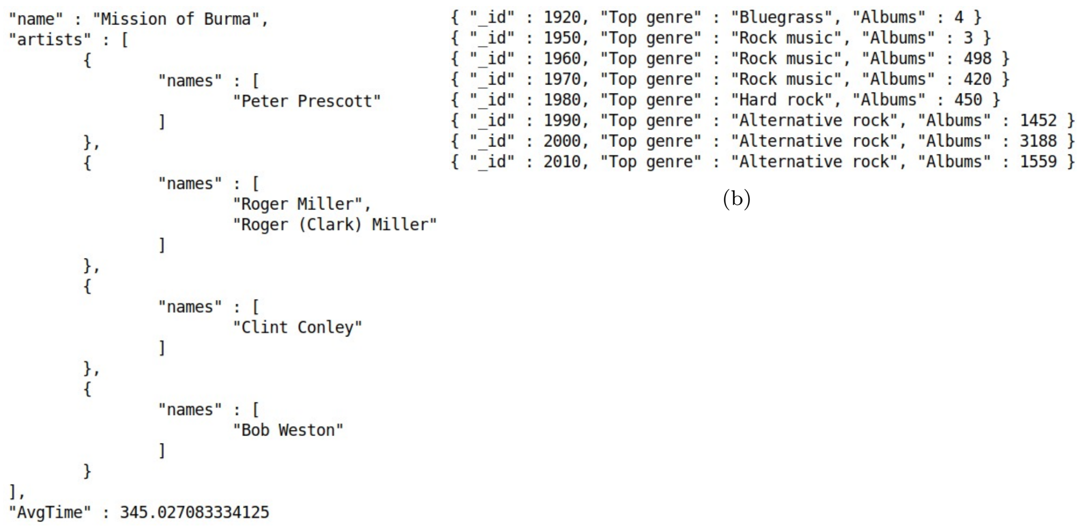
(a)



```
 Decade |       Genre       | Album_nr
--------+-------------------+----------
    192 | Bluegrass         |        4
    195 | Pop music         |        3
    195 | Rock music        |        3
    196 | Rock music        |      498
    197 | Rock music        |      420
    198 | Hard rock         |      450
    199 | Alternative rock  |     1452
    200 | Alternative rock  |     3188
    201 | Alternative rock  |     1559
```

(b)

Figure 4: (a) Results for the first query in SQL (b) Results for the second query in SQL



```
"name" : "Mission of Burma",
"artists" : [
        {
                "names" : [
                        "Peter Prescott"
                ]
        },
        {
                "names" : [
                        "Roger Miller",
                        "Roger (Clark) Miller"
                ]
        },
        {
                "names" : [
                        "Clint Conley"
                ]
        },
        {
                "names" : [
                        "Bob Weston"
                ]
        }
],
"AvgTime" : 345.027083334125
```

(a)

```
{ "_id" : 1920, "Top genre" : "Bluegrass", "Albums" : 4 }
{ "_id" : 1950, "Top genre" : "Rock music", "Albums" : 3 }
{ "_id" : 1960, "Top genre" : "Rock music", "Albums" : 498 }
{ "_id" : 1970, "Top genre" : "Rock music", "Albums" : 420 }
{ "_id" : 1980, "Top genre" : "Hard rock", "Albums" : 450 }
{ "_id" : 1990, "Top genre" : "Alternative rock", "Albums" : 1452 }
{ "_id" : 2000, "Top genre" : "Alternative rock", "Albums" : 3188 }
{ "_id" : 2010, "Top genre" : "Alternative rock", "Albums" : 1559 }
```

(b)

Figure 5: (a) Results for the first query in MongoDB (b) Results for the second query in MongoDB

5 Conclusion

Overall, both types of databases worked well for the data that was provided. Both types of databases have some advantages and disadvantages. Depending on what the intended purpose of the database is, either mongoDB or SQL can be the most suited. So if the goal is to explore the intricate relationships present in the structure of the data, probably SQL is a better choice, but if it's simply a matter of storage and accessing simple chunks of data, then mongoDB takes the cake.

Database implementation was easier on MongoDB, from deciding its structure (aggregate) to treating the data to be inserted into it, while in PostgreSQL there was a lot more discussion and it was rather unclear what would be the best way to organize the data.

In terms of querying, while Relational Databases share SQL language as a standard for querying, there is no declarative query language in the NoSQL world, making SQL queries supposedly easier to read and implement due to its english declarative model. MongoDB uses the MongoDB Query Language (MQL), designed for easy use by developers and has an aggregation framework to deal with more complex queries that is modeled on the concept of data processing pipelines: documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The first query was easier to write and perform in MongoDB, being mostly a match and sort operation, while in SQL the same query had 4 joins between tables. As for the second query, MQL revealed itself way less intuitive to use, mostly due to the query necessity to operate with arrays, grouping and decade boundaries that made the aggregation pipeline quite extensive. On the other side, SQL used 3 joins and a rank statement. It's very readable, making it easier to implement and debug. Based on the empirical evidence gathered from this project (there are only two examples, so it's hard to be too confident on this conclusion) we would say that perhaps for simple queries, mongoDB is easier to use, or at least as easy as SQL and as the query gets more complex, like in the second example, MongoDB becomes much harder to implement. The intuitive structure of SQL is the main factor for that difference.

The single aggregate chosen for MongoDB also influences the queries, making some of them easier (like query one) and others more difficult (like query two, because it has to dig into every aggregate).

Perhaps a graph database, such as Neo4J, could work better since it focuses on the intricate relationships established between all the elements. However, this type of database makes the insertion of data much more laborious.

References

[1] DBpedia. *DBpedia.* URL: `https://wiki.dbpedia.org/`.

[2] MongoDB. *MongoDB Documentation.* URL: `https://docs.mongodb.com/`.

[3] MongoDB. *MongoDB University.* URL: `https://pymongo.readthedocs.io/en/stable/`.

[4] MongoDB. *PyMongo 3.11.0 Documentation.* URL: `https://university.mongodb.com/`.

[5] PostgreSQL. *Documentation.* URL: `https://www.postgresql.org/docs/`.

[6] Raghu Ramakrishnan and Johannes Gehrke - 3^{rd}*edition*. *Database Management Systems.* McGraw-Hill, 2003. ISBN: 0072465638.

6 Appendix: Code for the creation of tables in PostgreSQL

```
CREATE TABLE Bands(
        band_id       SERIAL,
        band_URL      VARCHAR(150) NOT NULL,
        band_name     VARCHAR(70)  NOT NULL,
        PRIMARY KEY (band_id),
        UNIQUE(band_URL)
);


CREATE TABLE Genres(
        genre_id SERIAL,
        genre    VARCHAR(100)  NOT NULL,
        PRIMARY KEY(genre_id),
        UNIQUE(genre)
);


CREATE TABLE Bands_Genres(
        band_id     INTEGER ,
        genre_id    INTEGER,
        PRIMARY KEY(band_id,genre_id),
        FOREIGN KEY (band_id) REFERENCES Bands,
        FOREIGN KEY (genre_id) REFERENCES Genres
);

CREATE TABLE Artists(
        artist_id    SERIAL,
        artist_url   VARCHAR(150),
        PRIMARY KEY (artist_id),
        UNIQUE(artist_url)
);

CREATE TABLE Albums(
    album_id     SERIAL,
        band_id      INTEGER,
        album_name   VARCHAR(140) NOT NULL,
        release      DATE         NOT NULL,
        sales        INTEGER,
        time         FLOAT        NOT NULL,
        abstract     TEXT,
        PRIMARY KEY(album_id),
    UNIQUE(band_id,album_name,release,sales,time),
        FOREIGN KEY(band_id) REFERENCES Bands
);

CREATE TABLE Belongs(
        band_id         INTEGER,
        artist_id       INTEGER,
        artist_name     VARCHAR(150) NOT NULL,
        is_active       BOOLEAN      NOT NULL,
        PRIMARY KEY (band_id, artist_id, artist_name),
        FOREIGN KEY (artist_id) REFERENCES Artists,
        FOREIGN KEY (band_id) REFERENCES Bands
);

--copy the data

\copy Bands( band_URL, band_name) from '../new_sets/Bands.csv' DELIMITER ';' CSV HEADER;

\copy Genres( genre) from '../new_sets/Genres.csv' DELIMITER ';'  CSV HEADER;

\copy Bands_Genres(band_id, genre_id) from '../new_sets/Bands_Genres.csv' DELIMITER ';'
CSV HEADER;

\copy Artists(artist_url) from '../new_sets/Artists.csv'
DELIMITER ';' CSV HEADER;
```

```
\copy Albums(band_id, album_name, release, sales, time, abstract) from '../new_sets/Albums.csv'
DELIMITER ';' CSV HEADER;


\copy Belongs(band_id, artist_id, artist_name, is_active) from '../new_sets/Belongs.csv'
DELIMITER ';' CSV HEADER;
```

7 Appendix: JSON document format in MongoDB

```
{
"_id": ObjectId,
"url": String,
"name": String,
"genres": [genre_String_1, genre_String_2, ...],
"albums":[
                {"album_name": String,
                "release": Date,
                "sales": Int32,
                "time": Double,
                "abstract": String},

                {"album_name": String,
                "release": Date,
                "sales": Int32,
                "time": Double,
                "abstract": String}
        ],
"artists":[
                {"artist_url": String
                "names": [name_String_1, name_String_2, ...],
                "is_active": Boolean},

                {"artist_url": String
                "names": [name_String_1, name_String_2, ...],
                "is_active": Boolean}
        ]
}
```