Inf553 Project

Pierre Bourhis, Benoît Groz, Ioana Manolescu

October 2019

1 Overview

The ultimate goal of the project is to set up a Web application that enables users to exploit a music database, which is a reduced version of the MusicBrainz[1] dataset. The project is divided into two phases:

1. Create and load the database, express a set of queries in SQL, and try to improve their performance.
2. Write the Web application that allows to interact with the system through a Web browser.

2 Creating and loading the database

We provide the data and a skeleton of the relational schema. You have to:

- transform this into a set of table creations commands, with the necessary constraints;
- load the data in the tables thus created, using instructions we provide.

2.1 Data outline

The data contains artists, releases, and tracks, as well as additional information associated with them. A possible E-R diagram of the data appears in Figure 1.

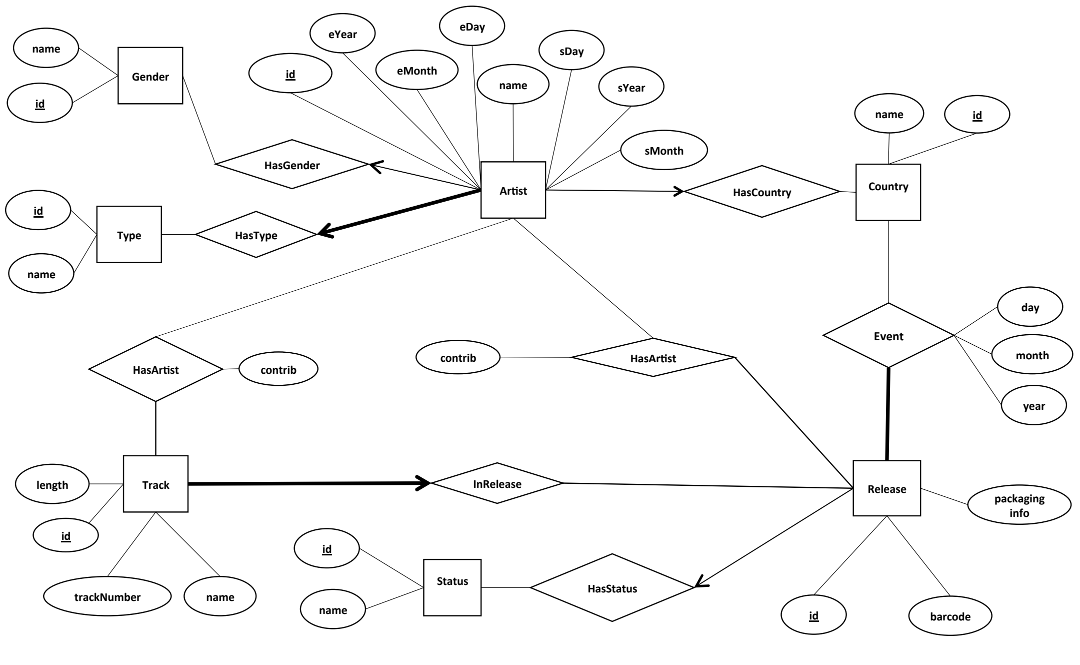

Figure 1: A possible Entity-Relationship model representing the music information.

[1] https://musicbrainz.org/

An artist is generally a musician, group of musicians, or other music professional involved in the production of a music release. An artist is not necessarily real, it can be also be a fictional character from a cartoon, movie, etc. (Daffy Duck, from Looney Tunes). Artists can be of one of the following types:

Person (an individual)	Group (a group of people)
Orchestra (a large instrumental ensemble)	Choir (a large vocal ensemble)
Character (fictional character)	Other

The database must support adding artists of any of these types.

Any artist has: a unique identifier, a name, a type (among those above), an associated geographic area, a start date and an end date. Artists of type Person also have a gender. The geographic area corresponds to the country of birth of a person, or the country where a group was formed; it may be unknown.

The start and end dates can mean different things depending on the type of the artist. Either they correspond to birth and death dates for individual persons, or the creation and end (dismantling) dates for groups and others. The start and end dates may be unknown, or they may be incomplete (e.g.,we know a birth year, but not the exact day and month).

A music release has: a release id, a title, a status, a barcode (max. 26 digits), and packaging information (max. 22 characters). The status, barcode, and packaging of a release may be unknown. If the status is known, it can take only one of the following values:

Official	Promotion	Bootleg	Pseudo-Release

Every release comprises work by one or more artists, and usually their contributions are not equal. For example, in the release "The Marshall Mathers"[2] the main contributor is Eminem, but other artists also participated in the release (e.g., Dido in the song "Stan"). Thus, apart from which artists participated in a release, the system must also store who is the first, second, etc., contributor, using an enumeration that starts from zero (where zero indicates the main contributor).

In practice, a release is often made *in several places* (a place is a country, or a set of countries) *at different times*, e.g. "released in US and the EU on Oct 1st, 2017; Japan release (of the same music) planned in Nov 2017". As these examples show, the date of release in a country (or group of countries) may be incomplete (missing day for the Japan release); it may also be unknown.

The various songs, and music compositions in a release, are stored as tracks. Each track is part of exactly one release. For every track we need to store the track unique identifier, the track name, the track number, the length (in milliseconds), and, when available, information about the artist(s) who contributed to each track. The rank of the contribution of each artist in each track must also be stored. As above (in releases), zero indicates the main contributor, and if more artists participated in a track, they are assigned increasing contribution numbers.

2.2 Relational schema

Extend the skeleton relational schema below into a full set of table creation statements, **without changing the table and attribute names**. You need to **add attribute types and constraints** as expressed in the E-R model and/or the above description.

```
artist_type(id, name)
gender(id, name)
country(id, name)
artist(id, name, gender, sday, smonth, syear, eday, emonth, eyear, type, area)
release_status(id, name)
release(id, title, status, barcode, packaging)
release_country(release, country, day, month, year)
release_has_artist(release, artist, contribution)
track(id, name, no, length, release)
track_has_artist(artist, track, contribution)
```

3 Loading the data

3.1 Data files

The project data can be found:

[2] https://en.wikipedia.org/wiki/The_Marshall_Mathers_LP

- in the directory `/users/misc-a/INF553-2019/ioana.manolescu/mbdump_reduced` accessible from any machine in the Polytechnique network. **If you work on a machine of Ecole polytechnique, it makes sense to load the data in your Postgres server from this directory, without copying it in your home directory (which would cause problems, because the data occupies 1.2 GB uncompressed).**

- otherwise (as a back-up), at the following address:
  `https://drive.google.com/file/d/0BwmEDVa0XUUlQ2xGY3B4RzBoTVE/view?usp=sharing` (chose the download option).
  If you download data from here, you need to decompress and unpack it by typing:

  ```
  tar -xvf mbdump_reduced.tar.gz
  ```

The data comes as a set of **10** CSV files, where each file corresponds to a relation of the relational schema. The largest relation (i.e.,tracks) has around **23 million** tuples and the smallest (i.e.,gender) only **3** tuples.

3.2 Constraint verification when loading data

The next sections discuss how to load efficiently data in a database. People sometimes speak of "bulk loading" for processes that load efficiently large amounts of data into the database. The general idea is to group instructions together and bypass some verifications, in order to avoid (most of) the overhead incurred when the DBMS performs individual INSERT operations.

Recall that a **transaction** is an atomic unit of change to a database. Either all the changes of a transactions are applied, or none is. The way this is implemented is: the changes that a transaction wants to make are applied in a temporary fashion[3], then, before deciding whether the changes should be made persistent, all the applicable integrity constraints are verified. Only if the constraints are still valid, the changes are made persistent (the actual database is modified). Supporting transactions has a big impact on the performance of database management systems. We explain below how we can curb that overhead when loading csv files.

3.3 Only for one or very few tuples: insert

We can use `insert` commands to insert tuple into a table. When this method is used, *each insertion is a transaction*. This means that the constraints are verified as many times as there are inserted tuples.

3.4 Standard loading from a file using copy

To load all the tuples from a file into a table, you can use a `copy` command, such as:

```
copy gender from '/users/misc-a/INF553-2019/ioana.manolescu/mbdump_reduced/gender';
```

This inserts all the tuples found in the file on the specified path, in the respective table. This is the usual way to load a database.

Such a `copy` command is a transaction. Thus, all the integrity constraints are verified only after all the tuples have been (temporarily) inserted, and just before the transaction ends. This is already much faster than tuple-by-tuple inserts! However, if one tuple insertion violates a constraint, the whole transaction is rejected (thus, no tuple inserted).

3.5 "Hack": efficient loading by avoiding constraint verification

To help you get the project done, we propose a "hack" which consists of *removing all constraints before loading the data, then re-declaring the constraints*. This way, we do not pay the price of verifying the constraints at insertion time! What is more, verifying constraints from scratch can be more efficient than verifying them incrementally when there are many modifications (as discussed in the lectures about B-trees).

Below, we list the steps for doing so.

1. Create your relational schema with the necessary constraints as required in Section 2.2.

2. Execute the following query whose result is **the commands needed in order to drop all the constraints from the database**:

[3] We will discuss the details of this later on in the course.

```
SELECT 'ALTER TABLE "'||nspname||'"."'||relname||'" DROP CONSTRAINT "'||conname||'" CASCADE;'
FROM pg_constraint
INNER JOIN pg_class ON conrelid=pg_class.oid
INNER JOIN pg_namespace ON pg_namespace.oid=pg_class.relnamespace
WHERE nspname = current_schema()
ORDER BY CASE WHEN contype='f' THEN 0 ELSE 1 END,contype,nspname,relname,conname;
```

Save the output of this query (we'll call it "the drop commands") in a file. **Do not run the drop commands (yet!)**

3. Execute the following query whose result is **a set of commands needed in order to re-create all the constraints**:

```
SELECT 'ALTER TABLE "'||nspname||'"."'||relname||'" ADD CONSTRAINT "'||conname||'" '
                     ||pg_get_constraintdef(pg_constraint.oid)||';'
FROM pg_constraint
INNER JOIN pg_class ON conrelid=pg_class.oid
INNER JOIN pg_namespace ON pg_namespace.oid=pg_class.relnamespace
WHERE nspname = current_schema()
ORDER BY CASE WHEN contype='f' THEN 0 ELSE 1 END DESC,contype DESC,nspname DESC,
         relname DESC,conname DESC;
```

Save the output of this query (we'll call it "the add commands") in a file.

4. Run the drop commands. They should all be successful; now your database has no constraints left.
5. Now load all the tables using `copy` commands as shown in Section 3.4. The insertion will basically be determined by the disk write speed only, as there are no constraints to check!
6. Run the commands that add the constraints to the database. Those constraints will be checked on the data you have added to the database, and checked again in case of further modifications to the database.

3.6 Obvious alternative

Create your database *without* the integrity constraints. Then load the data using copy commands.

Then issue `ALTER TABLE ... ADD CONSTRAINT...` statements you add all the necessary constraints to your database, as we requested in Section 2.2.

However, the smarter hack described in Section 3.5 can be applied to any database, even one previously created *with* constraints.

3.7 In case you need to drop the tables to restart

You may of course drop the database and re-create one. However, you may find it simpler to execute the following script which drops all tables and their depending objects.

```
DO $$ DECLARE
    r RECORD;
BEGIN
    -- in our labs, you do not need to know about the concept of PostgreSQL "schema"
    -- in short: system tables such as pg_tables belong to another "schema" in postgres terminology
    -- we do not want to try and delete these, hence the WHERE clause.
    FOR r IN (SELECT tablename FROM pg_tables WHERE schemaname = current_schema()) LOOP
        EXECUTE 'DROP TABLE IF EXISTS ' || quote_ident(r.tablename) || ' CASCADE';
    END LOOP;
END $$;
```

This script is not "pure SQL" but uses PL/pgSQL, the language for stored procedures in postgres. While standardized as an extension of SQL, the syntax for stored procedures differs slightly between systems. But there still are similar procedural languages in every major RDBMS. Technically, we could have used such a procedure above to drop the constraints in a single step, without resorting to copy-pasting. We believed it was easier to stick to the "pure" SQL that you are familiar with. Besides, we had to be careful about saving the instructions for restoring the constraints first.