Inf553 Project

Pierre Bourhis, Benoît Groz, Ioana Manolescu, Stamatis Zampetakis

October 2020

1 Overview

The ultimate goal of the project is to set up a *Web application* that enables users to exploit a *movie database*, which is a reduced version of a snapshopt of the IMDB[1] dataset. The project is divided into three phases:

1. Creating and loading the database, expressing a set of queries in SQL
2. Expressing a set of queries in SQL, and improving their performance.
3. Writing the Web application that allows to interact with the system through a Web browser.

The full set of files you need to work on is available as a **imdb-tables.zip** file here:
`https://drive.google.com/file/d/0B4t1oOFzubodRmZhV1NoOG1DWDQ/view`.

For your convenience, we also provide a **smaller** version of the same database, in which the largest tables have been scaled down (the other tables are the same). You can find them here:
`https://drive.google.com/file/d/1cqamuyPyg_xKpAGCcKQ8PCiOJZSfNlFS/view?usp=sharing`
Each file comprises the tuples of one relation; the relations, together, model the data from the Internet movie database (IMDB, in short). The smallest has 4 tuples; the largest has more than 5 million tuples. The data files have a total size of a few Gigabytes. Each file holds the tuples in csv (comma-separated values) format, with a header.

Both databases have the same schema, shape, constraints etc. Thus, the commands you write for the small one, will also work for the large one.

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 movies, as well as additional information associated with them:

- Each *movie* has an id (which is a key), a title, and a release year.
- The database also stores *actors*, *directors* and *producers*. Any of these has: an id (which is a key), a first name, a last name, and a birth date. Actors *play in* movies; directors *direct* movies; producers *produce* movies.
- A film may be known by more than one title, for instance, when the film title has changed in different language versions. There is no known upper bound on how many different titles can be thus associated to a film.
- A film may have zero or more ratings given to it by moviegoers. A rating consists of a number (value) between 1 and 5. A movie may have several ratings with the same value.
- Different kinds of information may be associated to a movie. For instance, a movie may have zero one or more awards, e.g., "Oscar in 2000" or "César in 1999". Famous quotes may be associated to a movie, e.g., "Of all the gin joints in all the towns in all the world, she walks into mine." Mistakes (inconsistencies) spotted in a movie may be added to the database... There is no fixed set of types of information that may be attached to a movie.

[1] `https://www.imdb.com/interfaces/`

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** and follow the description associated to each relation.

- `person (id, name, gender)`
  Person is a table with information about people involved in movies. It has the people's names, as in *Garbo, Greta*, and their gender.

- `info_type(id, info)`
  Info_type comprises about 100 different categories of information which may be attached *to a movie or to a person*. For instance, one info_type is *plot*: information items of this type are attached to movies. Another info_type is *birth date*: information items of this type are attached to people.

- `link_type(id, link)`
  Link_type comprises the categories of various links between movies, such as: movie m_1 is a remake of movie m_2, m_1 is similar to m_2 etc.

- `comp_cast_type(id, kind)`
  comp_cast_type is a table of four "codes" used to characterize *the type and quality of the movie cast information available within IMDB about a specific movie*. This table is a bit unusual. First, it contains a tuple *cast* and a tuple *crew*: cast is the set of people that appear in the movie, whereas crew is the set of all people who did something for the movie (the cast plus, e.g., the make-up artists, the music composer etc.) Second, it contains a tuple *complete* and a tuple *complete+verified*, to specify whether the cast information has been verified or not. Any cast (or crew) may be complete (or, complete and verified)[2].

- `movie_type(id, kind)`
  Movie_type comprises 7 categories of movies such as: movie, TV series, video game etc. Each movie is categorized in one of these types (movie has a foreign key referencing kind_type.id).

- `role_type(id, role);`
  Role_type comprises the categories of people that are involved in a movie: actors, producers, composers, costume designers etc.

- `aka_name(id, person_id, name)`
  Aka_name is a table of alternate names for people. This table has a foreign key on person.

- `movie(id, title, kind_id, production_year, episode_of_id, season_nr, episode_nr, series_years)`
  Movie table is the main table representing movies. It has the title, production year, episode number etc.

- `keyword(id, keyword)`
  Keyword is a table of keywords associated to the movies.

- `person_info(id, person_id, info_type_id, info, note)`
  Person_info stores information about people, much in the way movie_info has information about movies.

- `company(id, name, country_code)`
  Company has information about companies involved in the movies; companies have an associated company type (foreign key into company_type, see below).

- `company_type(id, kind)`
  Company_type comprises 18 roles that companies may play with respect to a movie: distributor, producer, special effects etc.

- `movie_company(id, movie_id, company_id, company_type_id, note)`
  Movie_company associates movies with companies: each tuple in this table has exactly two foreign keys, one into movies and the other into companies.

- `aka_title(id, movie_id, title, kind_id, production_year, episode_of_id, season_nr, episode_nr, note)`
  Aka_title is a table of alternative titles (some movies are known under more than one title). This table uses the attribute movie_id to encode which movie this is an alternative title for; movie_id is not a foreign key.

[2] In comp_cast_type, a single attribute is used to encode different things. While here we use the data "as-is", this is not, in general, good design.

- `char_name(id, name)`
  Char_name comprises movie character names.

- `cast_info(id, person_id, movie_id, person_role_id, note, role_id)`
  Cast_info contains associations between a person, a role, and a movie, to signify that the person played that role in that movie. Accordingly, each tuple in cast_info has a foreign key on a person, one on a movie, and one on char_name. Further, a cast_info tuple may also have another foreign key on role_type, to say what kind of role this was (actor, producer etc.) This table also has a field note, which, when not null, gives some extra information. For instance, in a tuple specifying that x was played the role of *director* in movie m, the field *note* may be used to say: "assistant: y" to denote that y was the assistant director. (The table role_type does not have an entry for assistant director.)

- `complete_cast(id, movie_id, subject_id, status_id)`
  Complete_cast characterizes the information available in the database about the complete cast of a certain movie. It has a foreign key to the movie, and: (i) one foreign key to comp_cast_type to specify whether the information available is about the *cast* or *crew*; and (ii) a second one to specify whether the cast is considered *complete* or *complete and verified*.

- `movie_rating(id, movie_id, info_type_id, info)`
  Movie_rating is a table that comprises rating information about movies. Rating information about a movie can be of up to five different types: (i) the average rating, (ii) the number of votes expressed on the movie, (iii) whether it is in the top 250 of IMDB, (iv) whether it is in the bottom 10 of IMDB, and (v) the detailed vote distribution[3].

- `movie_info(id, movie_id, info_type_id, info, note)`
  Movie_info has information about the movies, structured as follows: each tuple in movie_info is one piece of information about one movie. Thus, movie_info has a foreign key on movie.id to identify the movie. The information comprised in a movie_info tuple is in the info attribute. To enable interpreting this information (understanding what it is about), each movie_info tuple has another foreign key into info_type (see below).

- `movie_keyword(id, movie_id, keyword_id)`
  Movie_keyword associates movie titles with keywords (two foreign keys).

- `movie_link(id, movie_id, linked_movie_id, link_type_id)`
  Movie_link is a set of links between movies. Each tuple in movie_link references the first and second movie by means of foreign keys into movie.id; the fourth attribute is the type of the link (foreign key into link_type).

3 Loading the data

You are required to **create a set of tables**, to write the copy commands that **load a set of data files the tables** and to write a set of queries. This (large) database will serve for the future stages of the project.

Write down all your create/copy commands in a file and save them!

Work on the data schema as described in SubSection 2.1, with these exact table and attribute names, do not attempt to modify its organization. Also, you may find small errors or inconsistencies in the data; this is real data with real-world errors. Just use it as it is.

[3]Each detailed vote distribution information is *a string of length 10*, where each character may be either between 0 and 9, or '.' (dot), or '='. The character at position i, $1 \leq i \leq 10$, states which % of the ratings of the movie had value i (movies can be rated from 1 to 10). Thus:

- '.' specifies that 0% of the voters gave rating i;
- 0 specifies that at most 9% of the voters gave rating i;
- 1 specifies that at least 10% and less than 20% of the voters gave rating i and so on,
- 9 specifies that at least 90% but less than 100% of the voters gave the rating i,
- '=' specifies that 100% of the voters gave the rating i.

Constraint verification when loading data (read up to Section 3.4 before actually loading)!

Since the data is voluminous, it is important to load it efficiently. Recall that a *transaction* is an *atomic* set of changes to a database: either all the changes of a transactions are applied, or none is. As we will see later in the course, this is implemented as follows:

1. The changes that a transaction wants to make are applied in a temporary fashion;
2. Then, all the applicable integrity constraints are verified on the result of these temporary changes.
   - If there is no constraint violation, the changes are made persistent (the actual database is modified).
   - Otherwise, all the changes are rejected, and the database is left unchanged.

These checks can be expensive. We explain below how to limit their cost when loading the data.

3.1 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.

This would take a huge amount of time on the project dataset.

3.2 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 movie from '/path/to/my/files/movie.csv';
```

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 single 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).

This takes about half an hour on a 2017 laptop with SSD storage.

3.3 "Hack": efficient loading by avoiding constraint verification

To help you work faster, 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.

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**:

   ```
   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 result 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 (enter them in an interactive client, or call `psql commands.txt`). After running them, your database has no constraints left! You can check this by inspecting your tables, e.g., `\movie`.
5. Now load all the tables using `copy` commands as shown in Section 3.2. 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.

3.4 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 3.3 can be applied to any database, even one previously created *with* constraints.

Note: in these instructions, "actor" stands for "actress" as well as "actor".

4 Queries

In this step, you are required to **write a set of SQL queries** returning the following information over the IMDB database. Make sure no tuple appears several times in a query result.

1. Find the pairs of (movie title, actor name) such that the actor played in the movie[4].
2. Find the person names and movie titles such that the person has played in the movie a character called Amber.
3. Find the title and the year of production of each movie in which Nicolas Cage played. The lines for which the production year is unavailable should not be displayed. Order the results by year in descending order, then by title in ascending order.
   To test this query on the reduced database, you can try Kevin Ross instead of Nicolas Cage. We do expect the results on the large database.
4. Find the name of all the people that have played in a movie they directed, and order them by their names (increasing alphabetical order).
5. Find the titles of the movies that have only 1, 9 and 10 as rating.
6. Find the titles of the twenty movies having the largest number of directors and their number of directors, ordered by their number of directors in decreasing order.
7. Find the average number of episode Layla Covino played in, in her active years. A year is active if she plays in at least one movie produced that year. *For testing on the reduced database, use Kevin Ross.*

4.1 Updates

You can notice that the column `info` of the relation `movie_rating` can have some complex values when the info is a full value distribution. We propose to normalize this by creating a new relation `vote_distribution` having 11 attributes, one for the id and 10 for each each possible mark.

1. Create this relation.
2. Using an update command, migrate the information from `movie_rating` to the new relation.

[4]Return only the main names, not alternate names and titles.

5 What to turn in?

You should turn in through Moodle:

- the set of SQL commands used to deploy the schema of the database
- the set of SQL queries described in Section 3.7
- the set of SQL updates queries described in Sectin 3.8