哈尔滨工业大学

<<数据库系统>>

实验报告一

(2021 年度春季学期)

姓名：	李卓君
学号：	1180300210
学院：	计算机学院
教师：	邹兆年

1. 实验目的

1.学会正确运用概念数据库设计方法，正确使用实体-联系图(ER 图)表示概念数据模型。
2.学会正确运用逻辑数据库设计方法，在概念数据模型的基础上，设计合理的关系数据库模式。
3.学会正确运用物理数据库设计方法，根据工作负载，合理设计数据库的存取方法与存储结构。
4.掌握一种关系数据库管理系统(RDBMS)的使用方法，使用 SQL 创建、更新和查询关系数据库。
5.掌握数据库系统应用开发方法。

2. 实验环境

Windows 10 | Visual Studio Code | Python 3.8.5 | MySQL 8.0

3. 实验过程及结果

3.1 数据库设计

3.1.1 需求分析

在本实验中，我计划实现一个影评网站的后台数据管理系统。参照市场上现有的影评网站---豆瓣和 IMDB，一个影评网站的数据主要如下：

- 电影：包括标题、上映年份、制片地区、类型、评分、导演、演员表等属性，其中类似类型和制片地区等属性是多值的；
- 导演：包括姓名、性别、出生日期等属性；
- 演员：包括姓名、性别、出生日期等属性；
- 制片人：包括姓名、性别、出生日期等属性；
- 用户：包括姓名、性别、出生日期等属性；
- 片单：由用户创建，包括片单名字等属性；

这当中电影和导演、演员、制片人之间分别形成导演与由……导演、参演与由……参演、制片与由……制片的联系，参演还包括演员出演的角色属性，每部电影都要有导演、演员与制片人，而每个导演、演员与制片人都应该参与到至少一部电影的制作；用户与电影形成评论与被评论的联系，包括评论日期、评分和评论的复合属性；用户与片单之间形成创建与被创建、收藏与被收藏的联系；片单与电影之间形成列入与被列入的联系；而用户与用户之间还可以形成关注与粉丝的联系。

作为一个影评网站的后台管理系统，我计划实现以下功能：

- 电影的添加、修改、删除与查询；
- 演员的添加、修改、删除与查询，演员参与的电影与扮演角色的详细查询；
- 导演的添加、修改、删除与查询，导演参与的电影的详细查询；
- 用户的删除与查询，用户的关注、粉丝、创建的片单、收藏的片单、评论的详细查询；
- 片单的查询，片单包含的电影的详细查询。

对于一个影评网站，其主要存储的是用户的评论内容，后台管理系统的主要职责之一就是审查用户的评论是否违规，因此对于查询用户及其评论的性能有一定的要求。

3.1.2 概念数据库设计

根据 3.1.1 的需求分析设计的 ER 图见附图 1：db_lab1_ERgraph。

3.1.3 逻辑数据库设计

由 ER 模型转换的关系数据库模式如下：

- 实体型和 N:1 联系型转换成的关系数据库模式
  - Movie(movieID, title, year, avg_rating)
  - _User(userID, user_gender, user_name, user_birthdate)
  - Movie_List(mListID, list_name, createdby)
- 多值属性转换成的关系模式
  - genres(movieID, genre)
  - produce_countries(movieID, produce_country)
- M:N 联系型转换成的关系模式
  - Act_in(movieID, actor_name, actor_gender, actor_birthdate, role)
  - Direct(movieID, director_name, director_gender, director_birthdate)
  - Produce(movieID, producer_name, produce_gender, producer_birthdate)
  - Review(movieID, userID, date, numeric_rating, verbal_rating)
  - Listed(mListID, movieID)
  - _Like(userID, mListID)
  - Follow(userID, f_userID)

在 MySQL 中建立数据库后导出关系模式 ER 图见附图 2：MySQL_ERgraph。

3.1.4 物理数据库设计

3.1.2 给出的概念数据库设计和 3.1.3 给出的逻辑数据库设计是最终的数据库模式，最初的概念数据库设计 ER 图见附图 3：db_lab1_original_ERgraph。

可以看到，在最初设计的版本中，将电影的分类属性 Genre 设计为实体，该实体只有一个属性，主要是考虑到 Genre 在查询电影时可能会被经常使用，比较重要，所以单独为其设计实体，但在后来的设计中考虑到数据库应用更多时候进行的是嵌套查询，作为多值属性可能更好，而且可以在转换关系模式时精简一个表的设计。

最初设计的 Actor、Director、Producer 都是普通实体，各拥有主键 ID，同样在后续的转换关系模式时发现，这些实体与 Movie 组成的联系都是 M:N 的，如果按照弱实体处理可以各精简一个表的设计，并且精简后对于功能实现的影响极小。

建立索引方面，正如 3.1.1 需求分析中所述，对于用户及其评论的查询比较重要且频繁，因此对于用户的昵称增加了二级索引。除此以外，考虑到一些可能在未来数据库应用开发中可能用到的 SQL 语句操作，建立了如下二级索引：

- Movie:
  - KEY(avg_rating, title DESC)
- Movie_List:
  - KEY(createdby, list_name DESC)
- Review:
  - KEY(movieID, `date` DESC)
  - KEY(movieID, numeric_rating, verbal_rating(10) DESC)

3.1.5 数据库建立

建立数据库的 SQL 语句如下：

```
CREATE TABLE Movie(
            `movieID` int auto_increment,
            `title` varchar(255) NOT NULL,
            `year` year,
            `avg_rating` numeric(2,1),
            PRIMARY KEY(`movieID`),
            KEY (avg_rating, title DESC)
);

CREATE TABLE genres(
            `movieID` int,
            `genre` varchar(20),
            PRIMARY KEY(`movieID`, `genre`),
            FOREIGN KEY(`movieID`) REFERENCES Movie(`movieID`) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE produce_countries(
            `movieID` int,
      `produce_country` varchar(20),
      PRIMARY KEY(`movieID`, `produce_country`),
      FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE CASCADE
ON UPDATE CASCADE
);

CREATE TABLE _User(
```

```
        `userID` int auto_increment,
        `user_gender` ENUM('F', 'M', 'UNK') NOT NULL,
        `user_name` varchar(32) NOT NULL,
        `user_birthdate` date,
        PRIMARY KEY(userID), KEY (user_name)
);

CREATE TABLE Movie_List(
        `mListID` int auto_increment,
        `list_name` varchar(32) NOT NULL,
        `createdby` int,
        PRIMARY KEY(mListID),
        FOREIGN KEY (createdby) REFERENCES _User(userID) ON DELETE
CASCADE ON UPDATE CASCADE,
        KEY (createdby, list_name DESC)
);

CREATE TABLE Act_in(
        `movieID` int,
        `actor_name` varchar(64) NOT NULL,
        `actor_gender` ENUM('F', 'M'),
        `actor_birthdate` date,
        `role` varchar(64) NOT NULL,
        PRIMARY KEY(movieID, actor_name),
        FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE Direct(
        `movieID` int,
        `director_name` varchar(64) NOT NULL,
        `director_gender` ENUM('F', 'M') NOT NULL,
        `director_birthdate` date,
        PRIMARY KEY(movieID, director_name),
        FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE Produce(
        `movieID` int,
        `producer_name` varchar(64) NOT NULL,
        `producer_gender` ENUM('F', 'M') NOT NULL,
        `producer_birthdate` date,
        PRIMARY KEY(movieID, producer_name),
```

```
        FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE Review(
        `movieID` int,
        `userID` int,
        `date` date,
        `numeric_rating` int,
        `verbal_rating` text,
        PRIMARY KEY(movieID, userID),
        FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE
CASCADE,
        FOREIGN KEY(userID) REFERENCES _User(userID) ON DELETE CASCADE,
        KEY (movieID, `date` DESC),
        KEY (movieID, numeric_rating, verbal_rating(10) DESC)
);

CREATE TABLE Listed(
        `mListID` int,
        `movieID` int,
        PRIMARY KEY(mListID, movieID),
        FOREIGN KEY(mListID) REFERENCES Movie_List(mListID) ON DELETE
CASCADE ON UPDATE CASCADE,
        FOREIGN KEY(movieID) REFERENCES Movie(movieID) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE _Like(
        `userID` int,
        `mListID` int,
        PRIMARY KEY(userID, mListID),
        FOREIGN KEY(userID) REFERENCES _User(userID) ON DELETE CASCADE
ON UPDATE CASCADE,
        FOREIGN KEY(mListID) REFERENCES Movie_List(mListID) ON DELETE
CASCADE ON UPDATE CASCADE
);

CREATE TABLE Follow(
        `userID` int,
        `f_userID` int,
        PRIMARY KEY(userID, f_userID),
        FOREIGN KEY(userID) REFERENCES _User(userID) ON DELETE CASCADE
ON UPDATE CASCADE,
```

```
        FOREIGN KEY(f_userID) REFERENCES _User(userID) ON DELETE CASCADE
ON UPDATE CASCADE
);
```

索引已在上述代码中定义，下面是定义视图的语句：

```
CREATE VIEW movie_info AS SELECT * FROM Movie natural join genres natural join
produce_countries natural join Act_in natural join Direct;

CREATE VIEW director_movie AS SELECT * FROM Direct natural join Movie;

CREATE VIEW movie_detail AS SELECT * FROM Movie natural join genres natural join
produce_countries;

CREATE VIEW actor_movie AS SELECT * FROM Movie natural join Act_in;

CREATE VIEW user_follow AS SELECT * FROM _User natural join Follow;

CREATE VIEW user_mList AS SELECT * FROM _User join Movie_List WHERE userID =
createdby;

CREATE VIEW user_like_mList AS SELECT * FROM _User natural join _Like;

CREATE VIEW user_review AS SELECT * FROM _User natural join Review natural join
Movie;

CREATE VIEW movie_mList AS SELECT * FROM Movie natural join Listed;
```

3.2 数据库应用开发

3.2.1 数据库应用界面设计与实现

数据库应用的界面截图如图 3.1，现对于其各控件设计进行说明。

最上方的查询栏可以选择要查询的实体进行查询，可供查询的实体有电影、演员、导演、用户和片单，点按“选择”按钮进行选择。

第二栏的详细查询栏用于输入查询条件，对于图 3.1 中的电影查询，可以选择影片的种类、出产地区进行查询，同时也可以输入影片名、导演名、参演演员名进行查询。筛选好查询条件后，点按“查询”按钮进行查询。

第三行的结果栏用于展示查询结果，右侧有滚动条可进行下拉与上拉翻看结果。

最后一行为额外的功能栏，以支持增加、修改、删除等功能，点按结果栏中任意一行进行选择，再点按功能栏按钮即可完成操作。

同样的，以图 3.1 中的电影查询为例，选中电影编号为 1 的电影”The

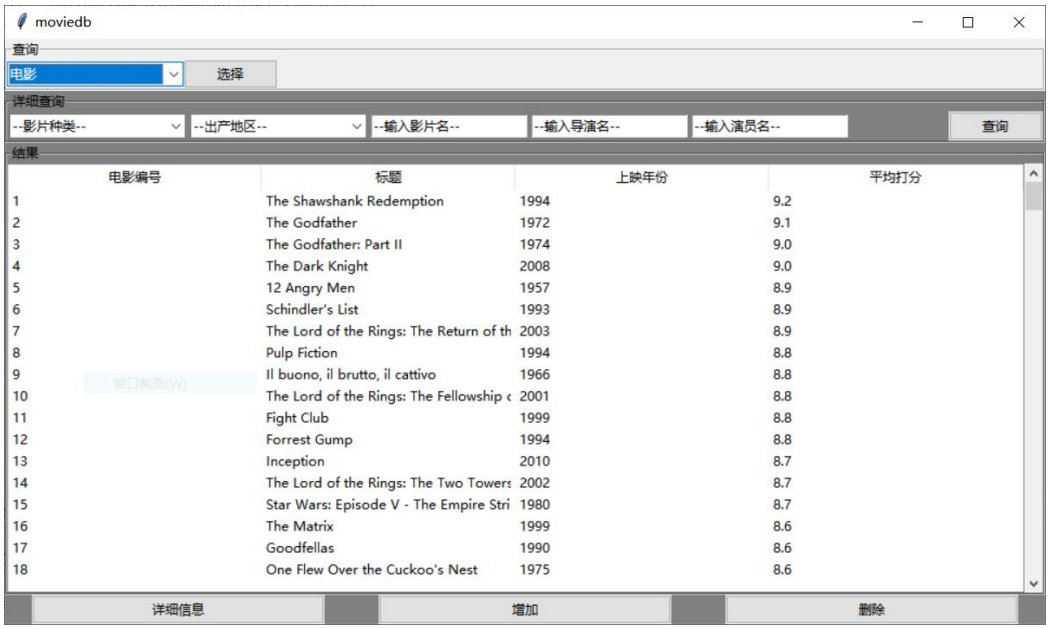


图 3.1

Shawshank Redemption”进行详细信息查询，其详细信息显示在新窗口如图 3.2 所示：

图 3.2

每个输入框的内容都可以进行修改，点按“保存更改”按钮进行保存，对于演职员

表的内容，需要先选中一行，点按“选择行”，该行的演职员信息就会被显示到输入框中，修改后点按“修改行”按钮进行修改。同样的，可以在选中演职员表中一行后，点按“删除演员”进行删除，或点按“增加演员”按钮为演职员表增加一行，默认的“演员”列会输入“演员名字”，“扮演角色”列会输入“扮演角色”。

对于其他实体的操作大同小异，这里仅以对于“电影”的操作为例，不再赘述。需要另外注意的是，为了符合一般情况下影评网站后台数据管理的要求，电影、导演和演员实体可以进行增删改查操作，其中因为要满足每部电影都有导演和演员参与，每个导演或演员都要参与到至少一部电影的制作中，所以导演和演员的增加操作通过电影页面实现，而后台对用户只能进行查询和删除操作，对于用户创建的片单只能进行查询操作。

数据库应用界面的实现代码在 ui.py 中，使用的为 python 自带的 tkinter 库，布局主要使用的为 grid 布局。

3.2.2 数据库应用功能设计与实现

数据库应用的功能实现代码在 function.py 文件中，主要实现的功能函数如下：

1.增加一部电影 insert_movie:

该函数接受 title, year, rating, director, genre, region 参数，首先利用 title, year, rating 向 Movie 表中插入一条电影，再通过在 Movie 表查询 title 获得该电影的 movieID，利用 movieID，插入多值属性 genre 和 region 以及外键参照 Movie 的 director。对于增加演员另外有函数 insert_actor；

2.增加一个演员 insert_actor:

该函数接受 movieID, actor_role 参数，向 Act_in 表中插入一条数据；

3.删除一个用户 delete_user:

该函数接受 userID 参数，从_User 表中删除一条数据；

4.删除一部电影 delete_movie:

该函数接受 movieID 参数，从 Movie 表中删除一条数据；

5.删除一名导演 delete_director:

该函数接受 director_name 参数，从 Direct 表中删除一条数据；

6.删除一个演员 delete_actor:

该函数接受 actor_name 参数，从 Act_in 表中删除一条数据；

7.修改一部电影 update_movie:

该函数接受 movieID, title, year, rating, director, genre, region 参数，对于 title, year, rating, director 属性直接根据 movieID 定位它们在表中对应的属性，使用 UPDATE 语句进行更新，对于多值属性 genre 和 region，由于不知道哪些属性遭到更改，哪些属性应该保留，属性是否减少或增加，因此不能简单地进行 UPDATE，需要先 DELETE 它们表中与 movieID 的对应项，再将新传入的参数进行 INSERT 操作；

8.修改一名导演 update_director:

该函数接受 ori_name, director_name, director_gender, director_birth 参数，更新 Direct 中的一条数据；

9.修改一个演员扮演角色 update_actor_role:

该函数接受 movieID, actor_role，actor_role 表示一个演员-角色对，是一个 list，和 update_movie 中的 genre, region 的更新一样，需要先删除再根据 movieID 添加

数据；

10.修改一个演员个人信息 update_actor：

该函数接受 ori_name, actor_name, actor_gender, actor_birth 参数，更新 Act_in 中的一条数据；

11.查询一部电影 select_movie：

该函数接受 movie_info 参数，movie_info 是一个 dict，取出对应键的值后，在视图 movie_info 中进行查询，查询利用字符串匹配 Like 功能，实现模糊查询；

12.查询一部电影的详细信息 get_movie_details:

该函数接受 movieID 参数，根据 movieID 在视图 movie_info 中进行查询；

13.查询一名导演 select_director:

该函数接受 director_name 参数，根据 director_name 在表 Direct 中进行查询；

14.查询一名导演的详细信息 get_director_details:

该函数接受 director_name 参数，根据 director_name 在表 Direct 和视图 movie_detail 中进行查询；

15.查询一个演员 select_actor：

该函数接受 actor_name 参数，根据 actor_name 在表 Act_in 中进行查询；

16.查询一个演员的详细信息 get_actor_details:

该函数接受 actor_name 参数，根据 actor_name 在视图 actor_movie 中进行查询；

17.查询一个用户 select_user:

该函数接受 user_name 参数，根据 user_name 在表_User 中进行查询；

18.查询一个用户的详细信息 get_user_details:

该函数接受 userID 参数，根据 userID 在表_User 和视图 user_follow, user_mList, user_like_mList, user_review 中进行查询；

19.查询一个片单 select_mList:

该函数接受 mList_name 参数，根据 mList_name 在视图 Movie_List 中进行查询；

20.查询一个片单的详细信息 get_mList_details:

该函数接受 mListID 参数，根据 mListID 在视图 Movie_List, _User, movie_mList 中进行查询；

3.2.3 数据库数据导入与功能测试

电影、导演、演员的部分数据来自 IMDB 网站前 250 榜单，爬取网页内容的代码见 crawl_imdb.py，代码仅供参考，只是单纯地将爬取内容经过处理输出到控制台上以供复制粘贴，未经规范化处理，不能保证运行效果。

完全爬取 IMDB 上所有电影内容或爬取导演、演员的所有信息的工作量较大，因此，电影只爬取了标题、类型、上映年份、制片地区、平均分数等内容，导演仅爬取了姓名属性，演员表仅爬取了每部电影的前三名演员的姓名属性，导演的性别全部设置为男，出生日期均设为 1970-1-1，演员的性别全部设置为女，出生日期均设为 1977-4-1。

用户和片单数据完全为代码生成的，最开始设置了 160 名用户，用户名为《百家姓》前 16 个的拼音加 0-9 的数字，用户性别对应其 ID 对 3 取模得到的数字对应的性别字符串，分别为“F”,“M”,“UNK”，用户的出生日期均为 1984-4-9。每名

用户与自己形成关注-粉丝关系，每名用户都创建并收藏一个片单，片单名称为用户名加字符串”favourite”，因此共有 160 个片单，每个片单各收藏一部电影。

后来为了体现增加索引带来的查询效率的提升，设置用户数为 160000 名，用户名为《百家姓》前 16 个的拼音加 0-9999 的数字，用户的其他属性与上述相同。上述代码见 generate_data.py，同样不能保证运行效果。

在 3.2.1 中已经展现了数据库应用的界面以及操作方法，先对于增加索引的效果加以介绍。

在总数为 160000 的用户中查询用户名为“feng999”，未加索引的查询时间为 0.123492s，见图 3.3。

增加索引后，进行同样的查询，查询时间为 0.00535625s，速度提升了 20 倍以上，见图 3.4 和 3.5。

```
MySQL 127.0.0.1:33060+ ssl db_lab1 SQL > select * from _User WHERE user_name = 'feng999';
+--------+-------------+-----------+----------------+
| userID | user_gender | user_name | user_birthdate |
+--------+-------------+-----------+----------------+
|  81000 | F           | feng999   | 1984-04-09     |
+--------+-------------+-----------+----------------+
1 row in set (0.1237 sec)
MySQL 127.0.0.1:33060+ ssl db_lab1 SQL > show profiles;
+----------+------------+-------------------------------------------------+
| Query_ID | Duration   | Query                                           |
+----------+------------+-------------------------------------------------+
|        1 | 0.00007175 | SHOW WARNINGS                                   |
|        2 |   0.123492 | select * from _User WHERE user_name = 'feng999' |
+----------+------------+-------------------------------------------------+
2 rows in set, 1 warning (0.0068 sec)
```

图 3.3

```
MySQL 127.0.0.1:33060+ ssl db_lab1 SQL > select * from _User WHERE user_name = 'feng999';
+--------+-------------+-----------+----------------+
| userID | user_gender | user_name | user_birthdate |
+--------+-------------+-----------+----------------+
|  81000 | F           | feng999   | 1984-04-09     |
+--------+-------------+-----------+----------------+
1 row in set (0.0056 sec)
```

图 3.4

```
143 | 0.00535625 | select * from _User WHERE user_name = 'feng999'
```

图 3.5

4. 实验心得

本次实验的工作量很大，但对于数据库操作的 SQL 语句的编写很少，难度也不高，实验的主要难度首先在于从零开始进行选题和ER 图的设计（上一次这样做还是在软件构造的课程中），在实现关系数据库模式和编写后端代码的过程

中需要根据需求不断来调整是造成这部分难度的主要原因。其次就是前端界面的编写，GUI 界面我选择使用了 python 自带的 tkinter 库，虽然最近的编译原理课程中也用 tkinter 参照一个开源的文本编辑器代码实现了一个简单的 GUI 词法分析器界面，但是其中有很多原理都没有弄明白。所以在本次实验中几乎也是从零开始学习 tkinter 各个部件的使用，还好网络上关于 tkinter 的博客很多，在解决一个个问题的过程中也算是收获良多了。当然，因为代码的不够规范，导致前后变量名不一致，以及 ER 图的设计并不是很完美，在实现过程中也造成了不少的工作量。

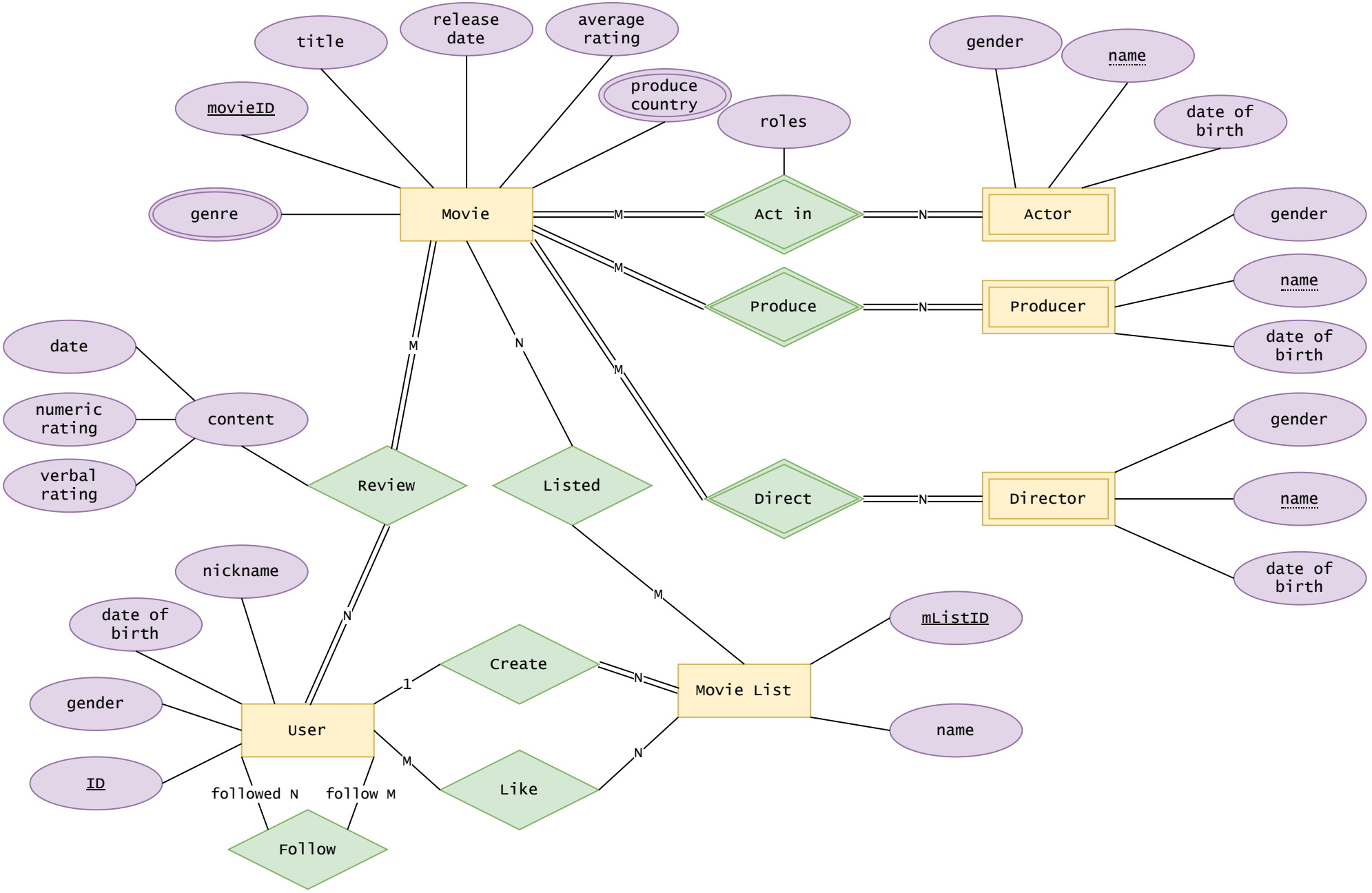
5. 参考资料

[1] Analyzing IMDb's Top 250 movies: Part 1; Let scrape some data . https://medium.com/analytics-vidhya/analyzing-imdbs-top-250-movies-part-1-let-scrape-some-data-a422adc3eb8d

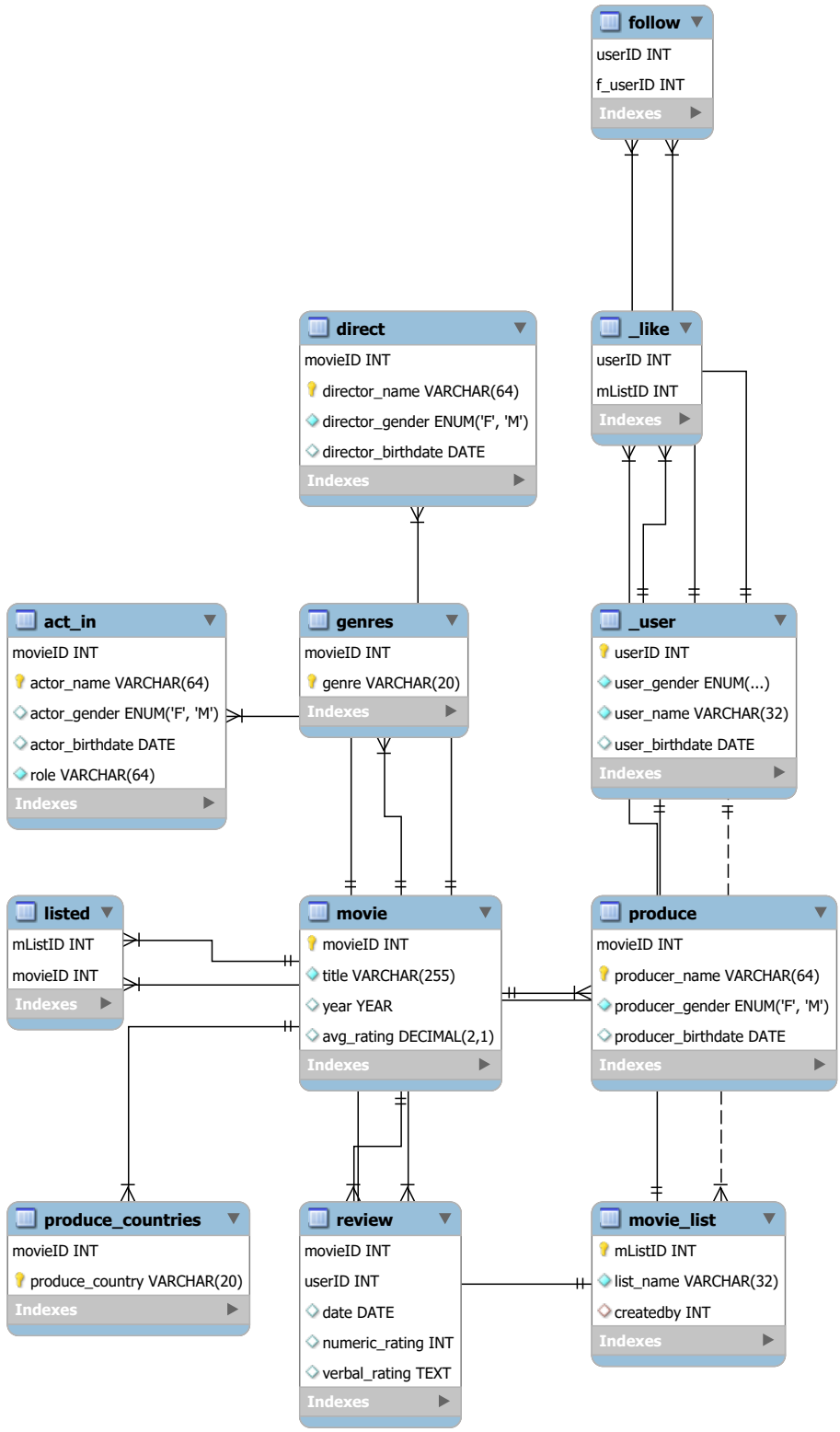
[2] Modeling Graph Database Schema. Noa Roy-Hubara, Lior Rokach, Bracha Shapira, Per etz Shoval. IEEE IT Professional 17.

[3] IMDB Top 250 Movies Details - Web scraping. https://www.kaggle.com/radrames/imdb-top-250-movies-details-web-scraping

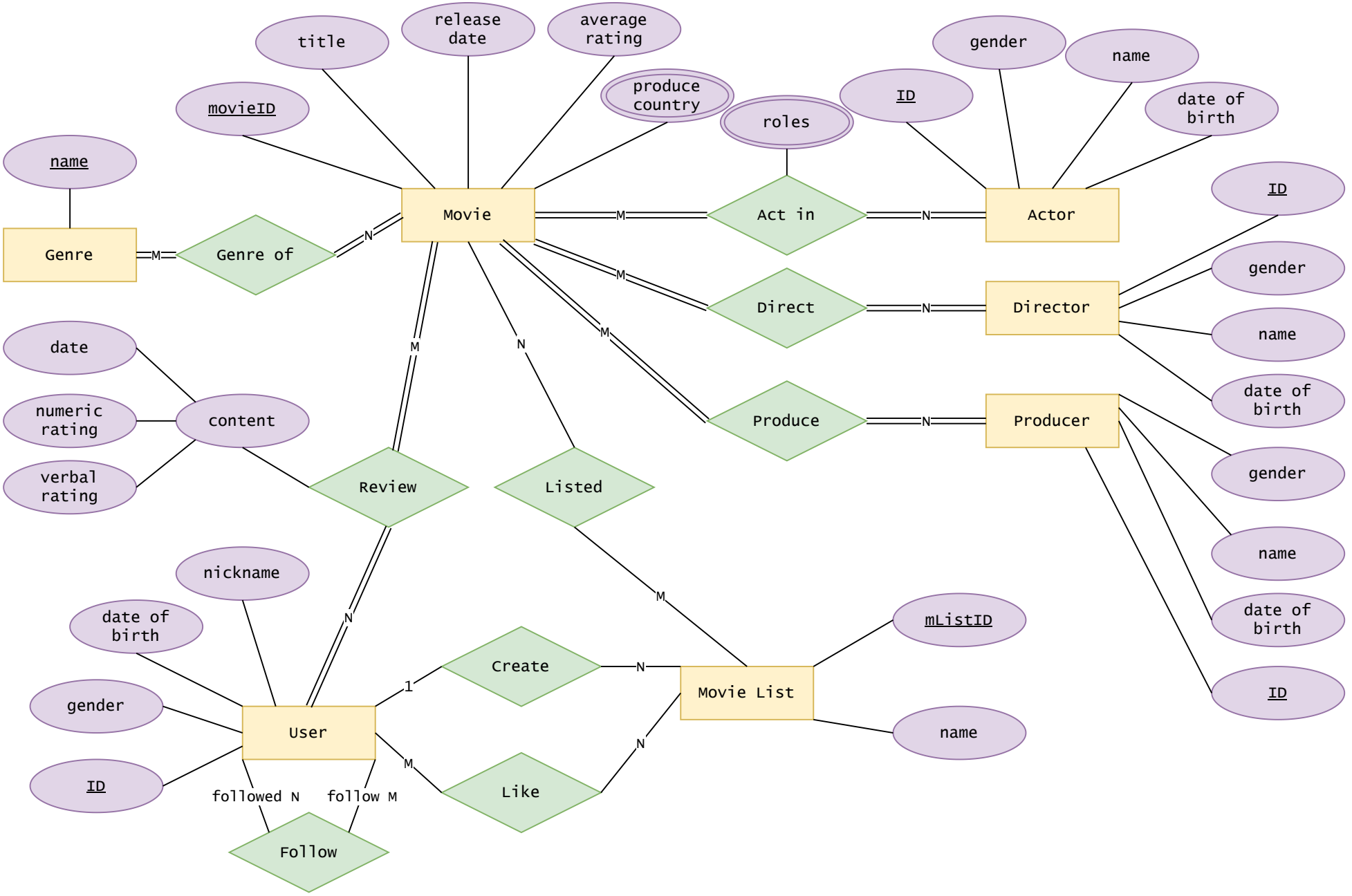
[4] tkinter — Python interface to Tcl/Tk. https://docs.python.org/3/library/tkinter.html



附图1：db_lab1_ERgraph



附图2： MySQL_ERgraph



附图3：db_lab1_original_ERgraph