Blavatnik School of Computer Science
Tel Aviv University

Database Systems Course

0368.3458

Home Work 3 – Movies Web App

Software Docs

By:
Sidney Feiner
Omer Moses

Contents

Software Documentation

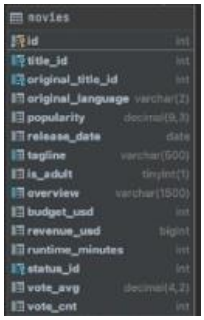
DB scheme structure

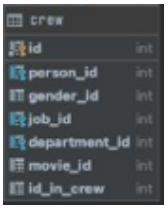
The design of the schema will be presented in several stages.

The key building blocks of our information are:

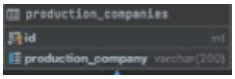
- movies
- crew
- cast

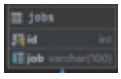
These building blocks contain the most basic information that needs to be used to generate more complex queries that incorporate information that users will want to know.





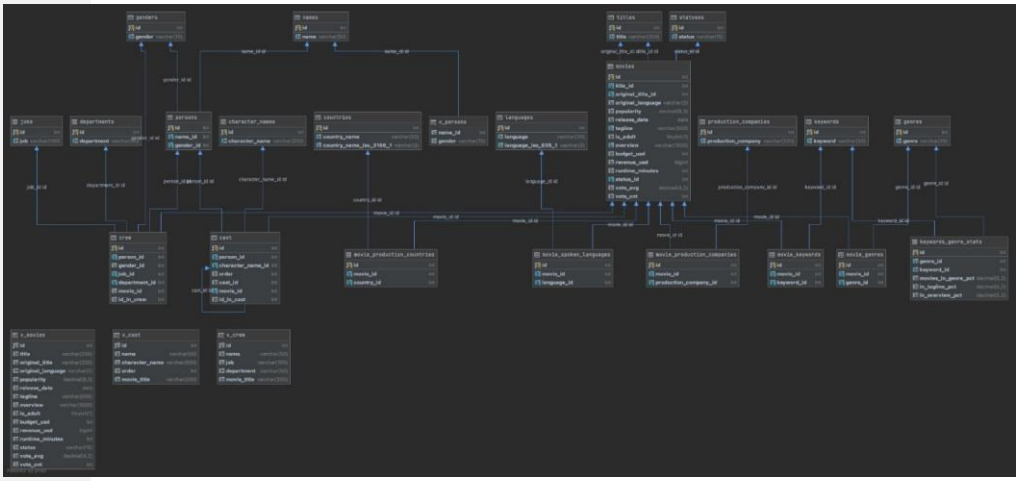
For information that does not change frequently, or that is not unique to a movie, actor, or team member, we have created dedicated tables. For example, if a staff member changes his department, and interested in getting offers in that new department, we can easily change that and it will be updated automatically. This way we are able to maintain scalable and efficient data updating.





Finally, connections between records are made based on key and foreign keys. So, for instance, if we take movie XXX his spoken language is stored in "movie_spoke_ languages" under some foreign key – language_id which is a key at "languages" table. Clearly, this information is not readable by the user. To solve this problem, we have created views that present the data in a textual way. Thus, for the case we presented, will be displayed to the user when executing a query, the English language for the movie XXX.

Our full DB schema:



DB optimization performed

In order to provide a positive user experience, along with optimal utilization of our resources, we have performed a number of DB optimizations:

1. Cron based data update – In order to keep our data fresh, we run an update across all of our data kept in our DB. Since the data isn't updated frequently, we will run a procedure that updates the data once a day. Note that during these procedures the user will have the ability to use the app based on old data, the data is updated atomically.
2. Indices – Our DB schema is build based on the features we provide to our users. After defining the features, we designed the schema to support those feature using indexes on data we would like to get easily. The following indices are applied:

Table Name	Index Over
names	name
keywords	keyword
character_names	character name

jobs	jobs
departments	department
genres	genre
titles	title
production_companies	Production companie
countries	country name country name iso
languages	language language iso
statuses	status

Queries Description

The following section describes our main implemented queries.

Get Movies

The query is an abstraction of a basic query pulling data from “movies” table.

Formally we can look at this query as

SELECT {projection choice}

FROM movies_table

WHERE {user choice}

Notes:

- Leaving the parameter projection choice empty is equal to SELECT *
- Leaving the user choice empty returns data from the table without conditioning

Optimization:

- Indices

How does the DB support this query:

- As mentioned earlier, movies table has a view the represents user readable data.
- Movies are the basis for data, and are used by us for the more complex queries built on the basis of these movies, the information is arranged in a separate basic table. In order to create an identification on the films they get a unique ID on the basis of which we can build more complex queries

Get Crew

The query is an abstraction of a basic query pulling data from “crew” table.

Formally we can look at this query as

SELECT {projection choice}

FROM crew_table
WHERE {user choice}

Notes:

- Leaving the parameter projection choice empty is equal to SELECT *
- Leaving the user choice empty returns data from the table without conditioning

Optimization:

- Indices

How does the DB support this query:

- As mentioned earlier, movies table has a view the represents user readable data.
- Crew is one of the basis for data, and used by us for the more complex queries built on this basis of these crew members, the information is arranged in a separate basic table. In order to create an identification on the members, they get a unique ID on the basis of which we can build more complex queries.

Get Cast

The query is an abstraction of a basic query pulling data from "cast" table.
Formally we can look at this query as

SELECT {projection choice}
FROM crew_table
WHERE {user choice}

Notes:

- Leaving the parameter projection choice empty is equal to SELECT *
- Leaving the user choice empty returns data from the table without conditioning

Optimization:

- Indices

How does the DB support this query:

- As mentioned earlier, movies table has a view the represents user readable data.
- Cast is one of the basis for data, and used by us for the more complex queries built on this basis of these crew members, the information is arranged in a separate basic table. In order to create an identification on

the members, they get a unique ID on the basis of which we can build more complex queries.

Multi Role Actors

In search for talented actors the query return actors that played multiple roles in a single movie.

Gender can be given and filtered by, but is optional.

```
select n.name, t.title, group_concat(distinct cn.character_name) as characters
from cast
join names n on cast.name_id = n.id
join movies m on cast.movie_id = m.id
join character_names cn on cast.character_name_id = cn.id
join titles t on m.original_title_id = t.id
{where_clause}
group by 1, 2
having count(distinct cn.character_name) >= 2
```

Optimization:

- Indices

How does the DB support this query: this query uses Cast and Movies tables and searches based on name_id which is a key in Cast table and movies_id and title which is foreign key from Movies table.

Lookalike Movies

The query takes a movie ID and returns lookalike movies. We defined lookalike movies are been graded based on

- Genre
- mutual crew/cast members

Eventually the query sorts the results based on the movie's reassembles score.

```sql
with same_genres as
         (
             select mg1.movie_id                                                as movie_id_1,
                    mg2.movie_id                                                as movie_id_2,
                    count(case when mg1.genre_id = mg2.genre_id then 1 end) as cnt
             from {contract.MOVIE_GENRES_TABLE} mg1
                      join movie_genres mg2 on mg1.movie_id <> mg2.movie_id
             where mg1.movie_id = %(movie_id)s
             group by 1, 2
         ),
     same_cast as (
         select c1.movie_id                                                as movie_id_1,
                c2.movie_id                                                as movie_id_2,
                count(case when c1.name_id = c2.name_id then 1 end) as cnt
         from {contract.CAST_TABLE} c1
                  join {contract.CAST_TABLE} c2 on c1.movie_id <> c2.movie_id
         where c1.movie_id = %(movie_id)s
         group by 1, 2
     ),
     same_crew as (
         select c1.movie_id                                                as movie_id_1,
                c2.movie_id                                                as movie_id_2,
                count(case when c1.name_id = c2.name_id then 1 end) as cnt
         from {contract.CREW_TABLE} c1
                  join {contract.CREW_TABLE} c2 on c1.movie_id <> c2.movie_id
         where c1.movie_id = %(movie_id)s
         group by 1, 2
     )
select m1.id    as requested_movie_id,
       t1.title as requested_movie,
       m2.id    as movie_id,
       t2.title as title,
       m2.original_language,
       sg.cnt   as mutual_genres_amt,
       sc.cnt   as mutual_cast_amt,
       scr.cnt  as mutual_crew_amt,
       m2.vote_avg,
       IF(m1.original_language = m2.original_language, 5, 0) + 3 * sg.cnt + 2 * sc.cnt +
       scr.cnt  as lookalike_score
from {contract.MOVIES_TABLE} m1
         join movies m2 on m1.id <> m2.id
         join same_genres sg on sg.movie_id_1 = m1.id and sg.movie_id_2 = m2.id
         join same_cast sc on sc.movie_id_1 = m1.id and sc.movie_id_2 = m2.id
         join same_crew scr on scr.movie_id_1 = m1.id and scr.movie_id_2 = m2.id
         left join titles t1 on m1.title_id = t1.id
         left join titles t2 on m2.title_id = t2.id
where m1.id = %(movie_id)s
order by lookalike_score desc,
         m2.vote_avg desc
```

Optimization:

- Indices

Index on movie_id in all relevant tables :עם הערות [SF1]

How does the DB support this query: the query searches based on genre that is indexed in his table for quicker search. The query also uses mutual cast members and crew workers that are primary keys in their table

Best Profit Members

The query takes a movie net revenue and divides it by the number of the entire workers who took place creating this movie (crew + actors).

```sql
with movie_members_amt as (
    select movie_id, count(distinct name_id) as members
    from (
             select movie_id, name_id
             from cast
             union all
             select movie_id, name_id
             from crew
         ) a
    group by 1
)
select m.id as movie_id, t.title as movie_name, (m.revenue_usd - m.budget_usd) / mm.members as profit_rate
from movies m
         join movie_members_amt mm on m.id = mm.movie_id
         join movie_genres mg on m.id = mg.movie_id
         join titles t on m.title_id = t.id
where genre_id = %s and m.revenue_usd is not null and m.budget_usd is not null
order by 1, profit_rate desc
```

Optimization:

- Indices

How does the DB support this query: Each movie registry in Movies table, has the net revenue value and this query gets the movie is as input. Next, movie_id is a forging key in both Crew and Cast table so we can use this index to search for relevant registries.

Loyal Crew Members

The query searches for crew members who worked for single production company – showing their loyalty to their company.

```sql
with crew_company as (
    select crew.name_id, crew.job_id, mpc.production_company_id
    from crew
             join movie_production_companies mpc on crew.movie_id = mpc.movie_id
)
select n.name, pc.production_company, group_concat(distinct jobs.job) as jobs
from crew_company cc1
         join names n on cc1.name_id = n.id
         join production_companies pc on cc1.production_company_id = pc.id
         join jobs on cc1.job_id = jobs.id
where exists(select 1 from crew where cc1.name_id = crew.name_id {jobs_filter})
    and not exists(select 1
                   from crew_company cc2
                   where cc1.name_id = cc2.name_id
                     and cc1.production_company_id <> cc2.production_company_id)
group by 1, 2
```

Optimization:

- Indices

How does the DB support this query: Each crew member registry in Crew table, holds a foreign key from movies_productions_companies called movie_id. Using this movie id we can easily find the correct movies registry and take the production company id which is a forging key in movies_productions_companies table.

Code Structure

CREATE-DB-SCRIPT

Contains the SQL queries we used to create the tables, tables and indices in our database.

API-DATA-RETRIEVE

This file is used to get the data from our API and csv files, parse it and propagate to our tables.

First, the system gathers the data and parse it from both of our sources.

Next, the data is being wrapped in a generator getting ready to be loaded into the database.

SERVER

This package contains services that responsible to:

- Querying the data at aql.py
- Exposing the API the our users will use while using the app at server.py

Description Of The API

Used Endpoints

We used themoviedb API to enrich our data base. Using rest API get call we get from the endpoint json formatted response with our DB's data.

First, we created a generator that iterates of movies data held in a dictionary. Using this generator we populate the data into the tables that we showed in our schema above.

Provided Endpoint

Our app users may choose to use any of our 9 provided endpoints. For the user's convenience, we provide a swagger that explains each endpoint, such as the type of operation, parameters, sample query and exceptions. You may use this link for opening the swagger guidelines.

/cast

Get Cast

The user my choose any of the following parameters he would like to know about the cast stored in our DB.

Note that leaving the parameters empty will return the all columns.

Parameter:

Name	Type	Description
ID	integer	ID of the member
name	string	Name of the member
character_name	string	character name of cast member in this movie

gender	string	gender
order	integer	order in the credits of cast member for this specific movie
movie_title	string	title of movie this member was cast in
limit	integer	If given, will limit the amount of returned records

Get Cast Multirole

Return actors that played multiple roles in a single movie. The user may choose one of the following parameters to filter the results getting from the app.
Parameter:

Name	Type	Description
Gender	string	gender
Limit	integer	If given, will limit the amount of returned records

/crew
Get Crew

The user my choose any of the following parameters he would like to know about the crew stored in our DB.
Note that leaving the parameters empty will return the all columns.
Parameter:

Name	Type	Description
ID	integer	ID of the member
Name	string	Name of the member
gender	string	gender
job	string	The Role of the member
department	string	Department of the member
movie_title	string	title of movie this member was cast in
limit	integer	If given, will limit the amount of returned records

/misc
Get Best Profit Per Worker

For a given genre, this endpoint returns the movie with the best profit per worker, where best profit per worker is the result of dividing the net revenue of the movie by all workers (cast + crew). Eventually, the result is movies where every worker had in the

average, the biggest contribution to the movie's profit.
Parameter:

Name	Type	Description
genre	string	ID of the member
limit	integer	If given, will limit the amount of returned records

Get Genre Distribution

Returns the distribution of genres for a given query. If no query is given, all movies will be taken into account.
Parameter:

Name	Type	Description
genre	enum	Use BOOLEAN QUERY syntax, which is described here: https://dev.mysql.com/doc/refman/8.0/en/fulltext-boolean.html

Get Keyword Genre Stats

Given genres, returns the most common keywords for these genres, with extra data regarding in how many movies of this genre (in percentage), the keyword appears in its description (either tagline or overview). More than limit keywords can be returned per genre, if some of these keywords have the same weight in the genre.
Note that leaving the parameters empty will return the all columns.
Parameter:

Name	Type	Description
ID	integer	ID of the member
name	string	Name of the member
gender	string	gender
job	string	The Role of the member
department	string	Department of the member
Movie Title	string	title of movie this member was cast in
limit	integer	If given, will limit the amount of returned records

Get Loyal Crew Members

The user may find crew members that work only with a single production company. API returns data for a specific job (or jobs). If no job is given, all jobs will be searched.
Parameter:

Name	Type	Description

job	array of comma separated strings	List of jobs
limit	integer	If given, will limit the amount of returned records

/movies

Get Movies

The user my choose any of the following parameters he would like to know about the cast stored in our DB.

Note that leaving the parameters empty will return the all columns.

Parameter:

Name	Type	Description
id	integer	ID of the movie
title	string	Name of the movie
original_language	string	
popularity	float	
release_date	string	Formatted as yyyy-mm-dd
tagline	string	
overviews	string	
budget_usd	integer	
revenue_usd	integer	
runtime_minutes	integer	
statuss	string	
vote_avg	float	On a scale of 0-10
vote_cnt	int	
projection	string	array of columns that we want the API to return. Can be an array containing any of the above columns. If no projection is given, all columns will be returned
limit	integer	If given, will limit the amount of returned records

Get Movies Lookalike

This query takes a movie ID and returns lookalike movies.

We defined lookalike movies are been graded based on

- Genre
- mutual crew/cast members

Results are sorted by the movie's reassembles score.

parameters:

Name	Type	Description
id	integer	ID of the member
name	string	Name of the member
gender	string	gender
job	string	The Role of the member
department	string	Department of the member
movie_title	string	title of movie this member was cast in
limit	integer	If given, will limit the amount of returned records

Opening The Swagger guidelines

Open the file SERVER/run-server.sh and insert correct user/password.

Once done, run this script and then the server will start running (on port 5050) (and you'll be able to use all our APIs) and in addition, you can see the whole API documentation at the endpoint: `SERVER:5050/apidocs/`.

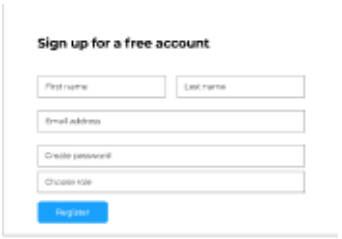
For example, if you're running locally: `http://localhost:5050/apidocs/`

General flow of the application

This section will provide three main flows that users experience while using the app. Both on the user's side and the app side we should know the role of the user in the cinema world, therefore the first flow we will registration. Next goes into out login flow where he should have successful identification, passing that, the landing page will be depended on the user's role. Eventually the user may move across our app features getting the data he need.

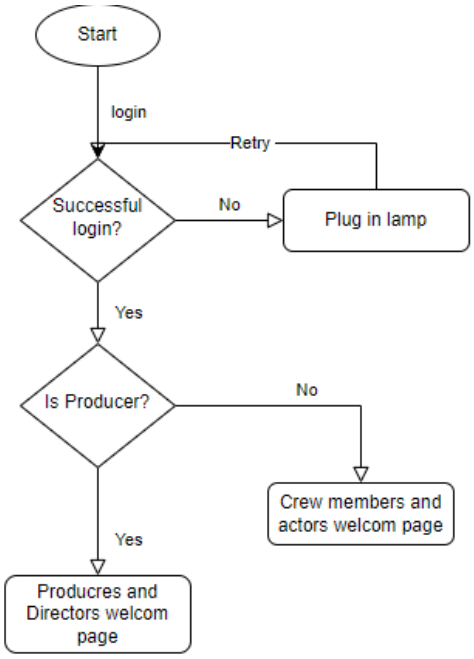
Registration Flow

The following form should be filled by the user. Note that this is a free account. As future plans we would like to create subscribers account:



Login Flow

The following basic flow chart describe the login flow stats machine.


Start
login
Retry
Successful login?
No
Plug in lamp
Yes
Is Producer?
No
Crew members and actors welcom page
Yes
Producres and Directors welcom page