DBMS Project Milestone 2

Team Name: 3 Amigos

Introduction

We did the following steps before starting to do querying :

- Data cleaning
- Created a bit representation for genres - Genre array of varchars has been converted to a single int. There are 29 genres in our db, the i-th bit of our int is set if genre array had the i-th genre. This saves tremendous amount of space and increase search speed a lot
- Exploding arrays - some of our data elements were arrays for which we created a different for each
- Creating the local database, importing from the cleaned data set
- Setting primary key constraints
- Quite disheartingly the column runtime Minutes had some string entries like 'Reality-TV', we removed all such entries

We have attached a file `cleaner.py` that we used to perform the above tasks

NOTE : Our database stores all the movies, shorts, videos, videoGames, Tvseries TVmovies etc. and every query will return one or some of these when a user searches. For the purpose of this report, whenever I talk about any of these objects I am going to use the term "movie" however please keep in mind that what might indeed by talked about can be a short or a tvseries as well.

Queries that will drive application

We have attached the queries that will be driving our application in the queries.sql file.

We basically thought of a large set of parameters that a user would like to set in order to filter and view movies of his choice. User has the option specify the following filters to obtain the search results :

1. **Substring Match** : Specify the name or part of a name of the movie, we then only show the movies that have that name occuring as a substring, irrespective of the case
2. **Type of content** : Specifying the type of content you are willing to watch, you can specify, movie, video, short, videoGame etc.
3. **Start Year** : Specify the constraint on start year for a series/movie
4. **Genre List** : Specify a list of genres from which you want to view the movies. Using the **cutOrIN** option you can specify if you want the movie to have all the genres or only some in your list
5. **Run time** : Specify the limit on runtime of the content you want to watch
6. **Episode Number, Season Number** : Specify the season and episode number if you want to watch a particular episode in a particular season, moreover you can also specify a range parameter for each of these
7. **Minimum and Maximum Episodes** : Specify if you want episodes from a series that has limits on the number of episodes. You don't want to start binge watching an anime with 1000+ episodes while at IITD !

There are many more generic options like these on the basis of which a user has the choice to filter contents from the database. We then build generic master queries that filter the relations on these basis and perform joins of these tables to pick desired results.This will help us integrating with back end where the backend can specify variables with values before running these generic queries. We then also support some special queries that a user may want to perform in over and above these or standalone. Some examples of such queries are :

1. **What are the top --k-- movies of all time**
2. **What are the top movies of this year**
3. **Recommend me some movies on the basis of my past ratings**
4. **What celebrities have the same age as me**
5. **What was the debut movie of a given actor**
6. **Movies in which an actor has done multiple roles**
7. **and many more ...**

We have included all these queries in 3Amigos.txt file

Index Choices for optimising Queries

Note that since we do not have permissions to create indices on tables on the uploaded database, we have constructed indices only on our local database and not on the dump that we have uploaded. We were receiving the following error :

```
group_12=> create index category_index on principals(category);
ERROR:  must be owner of table principals
```

Title

We have a lot of joins and searches on 'genre', this arises when we want to recommend a user a movie based on the genre he likes, in this case we want to search for a particular genre set in all of our title table. In this case if we have a secondary index on genre therefore this search becomes faster

```
create index genre_index on title(genres);

create index sy_index on title(startyear);

create index ey_index on title(endyear);
```

Very often we require to query our database on the basis that the startyear must be in a particular range or the end year must be in a particular range, or the episode number must be in a particular range. We know that when we have a btree index then range search becomes must faster, this is because leaves in a btree are present in a sorted order and each leaf has a pointer to the next leaf, once you have the start and end the limit on a search you essentially just have to do a linked list traversal from the start till end. We employ this property of btree indxeing to our advantage

We thus obtain the following final indices on title :

```
-------------+-------------------------------------------------------------
--
 title_pkey  | CREATE UNIQUE INDEX title_pkey ON title USING btree
(titleid)
 genre_index | CREATE INDEX genre_index ON title USING btree (genres)
 sy_index    | CREATE INDEX sy_index ON title USING btree (startyear)
 ey_index    | CREATE INDEX ey_index ON title USING btree (endyear)
(4 rows)
```

Persons

Similarly when we have queries on the age of an actor ie. when we want the age to be in a partcular range, we would want it to be btree indexed for faster search hence we create the above indices and obtain the following indices on person table :

```
create index dy_index on persons(deathyear);
create index by_index on persons(birthyear);
```

```
  indexname   |                                indexdef

--------------+-------------------------------------------------------------
--------
 persons_pkey | CREATE UNIQUE INDEX persons_pkey ON persons USING btree
(personid)
 dy_index     | CREATE INDEX dy_index ON persons USING btree (deathyear)
 by_index     | CREATE INDEX by_index ON persons USING btree (birthyear)
(3 rows)
```

Principals

We often have search on the category of a role a person played in a movie, so to enhance this search we create the following index on principal.

```
create index category_index on principals(category);
```

```
---------------+-------------------------------------------------------
-------------
 category_index | CREATE INDEX category_index ON public.principal USING
btree (category)
(1 row)
```

Ratings

On the rated table we have to search on the basis of rating value being more than some number , we also have to search on the tile that has been rated, so we create two indices for this table.

```
create index rating_title_index on rated(titleid);

create index rating_index on rated(rating);
```

```
     indexname      |                                indexdef

--------------------+-------------------------------------------------
-----------------
 rated_pkey         | CREATE UNIQUE INDEX rated_pkey ON rated USING btree
(userid, titleid)
 rating_index       | CREATE INDEX rating_index ON rated USING btree
(rating)
 rating_title_index | CREATE INDEX rating_title_index ON rated USING btree
(titleid)
(3 rows)
```

Episode

Whenver we need to count the number of episodes in a sries we need to group on the basis of a common parent, in this case an index on episode.parenttitleid is extremely helpful , hence we make one.

```
create index ep_parent on episode(parenttitleid);

create index ep_season on episode(seasonnumber);

create index ep_epnum on episode(episodenumber);
```

and finally we have the following indices on episodes table

```
  indexname   |                              indexdef

--------------+-------------------------------------------------------------
-------
 episode_pkey | CREATE UNIQUE INDEX episode_pkey ON episode USING btree
(titleid)
 ep_parent    | CREATE INDEX ep_parent ON episode USING btree
(parenttitleid)
 ep_season    | CREATE INDEX ep_season ON episode USING btree
(seasonnumber)
 ep_epnum     | CREATE INDEX ep_epnum ON episode USING btree
(episodenumber)
(4 rows)
```

personKnownForTitle

Finally for the table `personKnownForTitle` each entry is a tuple of 2 elements which is itself the primary key and hence contains a btree index already. Therefore there is no need to create any other index.

```
        indexname         |
indexdef
--------------------------+-------------------------------------------------
-------------------------------------------------
 personknownfortitle_pkey | CREATE UNIQUE INDEX personknownfortitle_pkey ON
personknownfortitle USING btree (personid, titleid)
(1 row)
```

Users and Rated

For these tables we have very less amount of data write now, so we have not constructed any indices on these tables since, once the data amount increases we can consider making some.

Database Size and Performance

We are using a large raw dataset that we obtained from https://datasets.imdbws.com/ .

We cleaned the data, did normalisation and division into tables of the required field and created our sql database with 8 public tables of the following sizes :

Table Name	Size (MB)	Number of records
principals	3513	55,183,051
personKnownForTitle	1878	21,601,830
title	1332	9,785,883
persons	1044	12,379,450
episode	633	7,358,491
ratings	113	1,291,165
users	0.032	18
rated	0.032	6

Note that the users and rated tables store the new users that we register and the ratings that they give to a particular title and hence they are small as of now, and will increase as more and more users are added and they start rating movies.

The time for each query has been mentioned in the .sql along with the what that query does

We see that the time to run queries on the database, we were expecting large speed ups if we could create indices but since that was not possible on the uploaded database, we could not speed up the queries a lot. Still we can see that due to appropriate primary indices on the table , the performance is good as per the size of our database, most of the large queries terminate in under 30s

As an example of the output of our queries, we present the output of one of our queries that find the top rated movie/series of all time .
the query is as follows :

```
From title t Join ratings r On (t.titleid = r.titleid and r.averageRating
is not NULL)
Order By (r.averageRating * r.numVotes) DESC
LIMIT 10;
```

You can see that we get the popularly best known movies and series of all time.

More over the time for this query is also just 1.874 s