**Team Debauchery**

Ariel Cohen: ariel4

Dirk Hortensius: dirker27

**Database Application Description**

Our database application, VTFlix, will act like a social IMDB where users can look up movies, rate movies, and look at ratings from friends.

**Project Webpage**

cs4604.cs.vt.edu/~dirker27

**Member Roles**

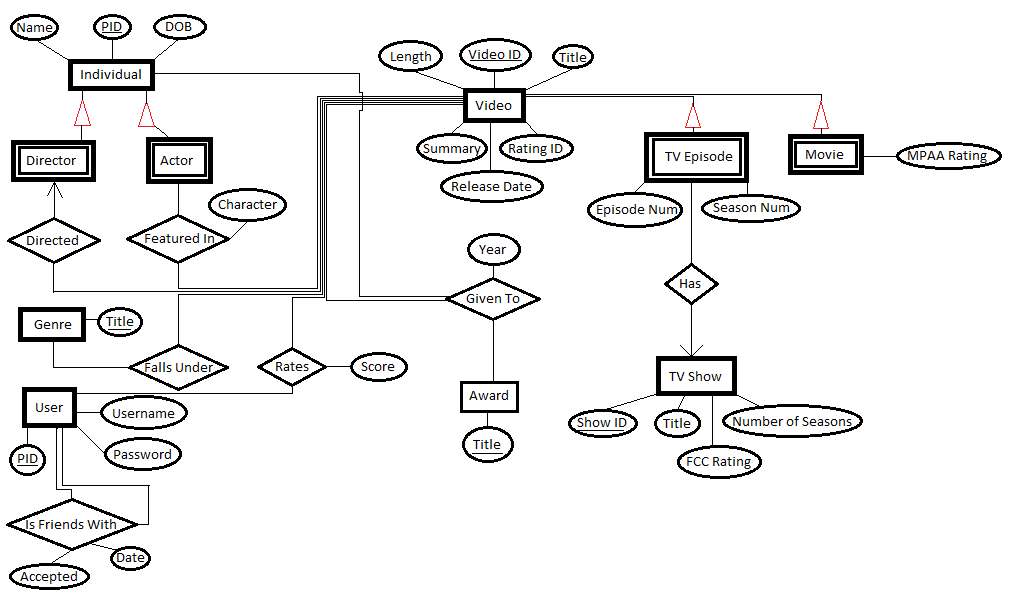
**Project 1**: Both designed the ER Schema together. Ariel wrote the entity and relationship descriptions while Dirk redrew the schema to look more professional. Both wrote half of the SQL DDL statements.

**Project 2**: Both answered the FD and BCNF/3NF questions together. Both agreed on the DDL statements used to create and query the database, then Dirk used the DDL statements and found the tuples and query results while Ariel typed up the report.

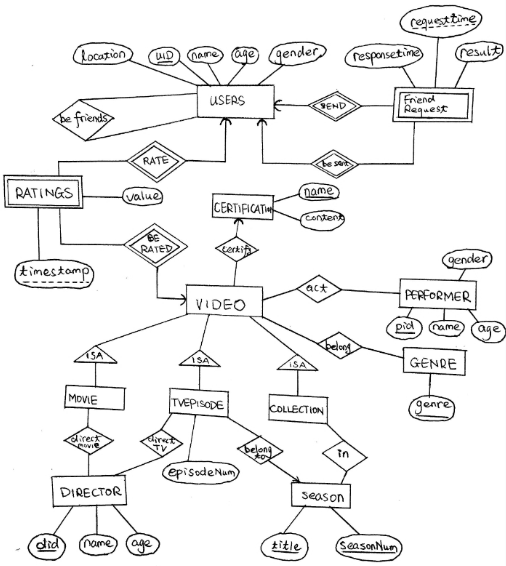
**Project 3**: Ariel put together the report and created the HTML/CSS files. Dirk worked with php and linking the database to the webpage.

**ER Schema**

Our original ER Schema in project 1 looked like:



However, from project 2 on, we conformed to the given ER Schema:



However, we did rename the entities and relations to what we found my inherent for us, as will be described in the following entity and relation descriptions.

**Entity Descriptions**

**User Info**

The user entity holds all information on the user. Information includes a user ID (uid), user age (age), user gender (gender), user’s first and last name, (first\_name, last\_name), and the user’s location (location).

**Performer**

The performer entity holds all information on the performer. Information includes a performer ID (pid), the performer’s first and last name (first\_name, last\_name), the performer’s age (age), and the performer’s gender (gender).

**Director**

The director entity holds all information on the director. Information includes a director ID (did), the director’s first and last name (first\_name, last\_name), and the director’s age (age).

**Certificates**

The certificate entity holds all information on a certificate. Information includes the name of the certificate (certification\_name) and the content a certificate holds (content).

**Genre**

The genre entity holds the name of the genre (genre).

**Video Info**

The video entity holds all information on a video. Information includes a video ID (vid), the title of the video (title), the release year of the video (release\_year), the name of the video’s produce (producer), the name of the country the video is from (country), and the name of the video’s certification that must also be in the certificates table (certification\_name).

**Movie Info**

The movie entity holds all the information for a movie. Information includes all video information (since a movie is a video) as well as a movie ID (mid) and a movie title (mtitle).

**TV Episode Info**

The TV episode entity holds all information for a TV episode. Information includes all video information (since a TV episode is a video) as well as an episode ID (tvid) and an episode number (episode\_num).

**Collection**

The collection entity holds all information for a collection. Information includes all video information (since a collection is a video) as well as a collection ID (collection\_id) and a name of the collection (collection\_name).

**Season**

The season entity holds all information for a season. Information includes a season title (season\_title) and a season number (season\_num).

**Relation Descriptions**

**Belongs To Genre**

The belongs to genre relation is a relationship between a video and a genre to represent a video belonging to a genre. A video can belong to one or more genres, creating a many-to-many relationship. This table grabs the primary keys from both the video (vid) and the genre (genre) and has no extra information.

**Ratings**

The ratings relation is between a video and a user to represent a user giving a video a rating. A user can rate multiple videos and videos can be rated by multiple users, creating a many-to-many relationship. This table grabs the primary keys from both video (vid) and the user (uid) and adds a rating score (rate\_score) and a time of the rating (ratingtime).

**Friend Requests**

The friend requests relation is between two users to represent a user requesting to be friends with another user. A user can request to be friends with multiple other users, creating a many-to-many relationship. This table grabs the primary key from both users (uid) and adds a time of the request (requesttime), a response (response), and a time of the response (responsetime).

**Friends**

The friends relation is between two users to represent two users being friends. A user can be friends with multiple other users, creating a many-to-many relationship. This table grabs the primary key from both users (uid) and adds no other information.

**Act In**

The act in relation is between a performer and a video to represent a performer acting in a video. An actor can be in multiple videos and a video can consist of multiple actors, creating a many-to-many relationship. This table grabs the primary key from the performer (pid) and the video (vid) and adds no other information.

**In Season**

The in season relation is between a TV episode and a season to represent a TV episode being in a particular season. A season holds multiple TV episodes, but a TV episode can only be in one season, creating a many-to-one relationship. This table grabs the primary keys from TV episode (tvid) and season (season\_title and season\_num) and adds no further information.

**In Collection**

The in collection relation is between a collection and a season to represent a season being in a particular collection. A collection can hold multiple seasons, but a season can only be in one collection, creating a many-to-one relationship. This table grabs the primary keys from collection (collection\_id) and season (season\_title and season\_num) and adds no further information.

**Movie Directors**

The movie directors relation is between a movie and a director to represent a director directing a movie. A director can direct multiple movies, but a movie can only have one director, creating a many-to-one relationship. This table grabs the primary keys from video (vid) and director (did) and adds no further information. We meant to pull the primary key from movie (mid) instead of video, but realized this mistake too late.

**TV Episode Directors**

The TV episode directors relation is between a TV episode and a director to represent a director directing a TV episode. A director can direct multiple TV episodes, but a TV episode can only have one director, creating a many-to-one relationship. This table grabs the primary keys from TV episode (tvid) and director (did) and adds no further information.

**Relational Schema**

Following the given information, we created the following relational schema of entity and relational tables. (Please again note that the movie directors schema is slightly incorrect – it should pull movie’s mid instead of video’s vid.)

**UserInfo**(uid INT PRIMARY KEY, age INT, gender CHAR, first\_name VARCHAR,

last\_name VARCHAR, location VARCHAR)

**Performer**(pid INT PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, age

INT, gender CHAR)

**Director**(did INT PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, age

INT)

**Certificates**(certification\_name VARCHAR PRIMARY KEY, content VARCHAR)

**Genre**(genre VARCHAR PRIMARY KEY)

**VideoInfo**(vid INT PRIMARY KEY, title VARCHAR, release\_year INT, producer

VARCHAR, color VARCHAR, country VARCHAR, certification\_name

VARCHAR REFERENCES Certificates(certification\_name))

**BelongToGenre**(vid INT REFERENCES VideoInfo(vid), genre VARCHAR

REFERENCES Genre(genre), PRIMARY KEY (vid, genre))

**MovieInfo**(mid INT PRIMARY KEY, mtitle VARCHAR)

**Ratings**(vid INT REFERENCES VideoInfo(vid), uid INT references UserInfo(uid),

rate\_score INT, ratingtime TIME, PRIMARY KEY(vid, uid, ratingtime))

**FriendRequests**(uid1 INT REFERENCES UserInfo(uid), uid2 INT REFERENCES

UserInfo(uid), requesttime TIME, response VARCHAR, responsetime TIME,

PRIMARY KEY(uid1, uid2, requesttime))

**Friends**(uid1 INT REFERENCES UserInfo(uid), uid2 INT REFERENCES

UserInfo(uid), PRIMARY KEY (uid1, uid2))

**ActIn**(vid INT REFERENCES VideoInfo(vid), pid INT REFERENCES Performer(pid),

PRIMARY KEY (vid, pid))

**TVEpisodeInfo**(tvid INT PRIMARY KEY REFERENCES VideoInfo(vid), episode\_num

INT)

**Collection**(collection\_id INT PRIMARY KEY REFERENCES VideoInfo(vid),

collection\_name VARCHAR)

**Season**(season\_title VARCHAR, season\_num INT, PRIMARY KEY(season\_title,

season\_num));

**InSeason**(tvid INT REFERENCES TVEpisode(tvid), season\_title VARCHAR,

season\_num INT, FOREIGN KEY (season\_title, season\_num) REFERENCES

Season(season\_title, season\_num), PRIMARY KEY(tvid, season\_title,

season\_num))

**InCollection**(collection\_id INT REFERENCES Collection(collection\_id), season\_title

VARCHAR, season\_num INT, PRIMARY KEY(collection\_id, season\_title,

season\_num), FOREIGN KEY(season\_title, season\_num) REFERENCES

Season(season\_title, season\_num))

**MovieDirectors**(did INT REFERENCES Director(did), vid INT REFERENCES

VideoInfo(vid), PRIMARY KEY (did, vid))

**TVEpisodeDirectors**(did INT REFERENCES Director(did), tvid INT REFERENCES

TVEpisodeInfo(tvid), PRIMARY KEY (did, tvid))

**Creating the Database**

Following the schema, we used the following DDL statements to create our database.

==================================================================

NOTE: All tables with initial information were renamed to PRAK\_<tablename> to differentiate between them and the final ones.

**USER INFO**

-----------------------------

CREATE TABLE UserInfo(uid INT PRIMARY KEY, age INT, gender CHAR, first\_name VARCHAR, last\_name VARCHAR, location VARCHAR);

INSERT INTO UserInfo SELECT DISTINCT uid, age, gender, first\_name, last\_name, location FROM Prak\_Users;

**PERFORMER**

-----------------------------

CREATE TABLE Performer(pid INT PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, age INT, gender CHAR);

INSERT INTO Performer SELECT DISTINCT pid, first\_name, last\_name, age, gender FROM Prak\_Act;

**DIRECTOR**

----------------------------

CREATE TABLE Director(did INT PRIMARY KEY, first\_name VARCHAR, last\_name VARCHAR, age INT);

INSERT INTO Director SELECT DISTINCT did, first\_name, last\_name, age FROM Prak\_VideosDirect;

**CERTIFICATES**

----------------------------

CREATE TABLE Certificates(certification\_name VARCHAR PRIMARY KEY, content VARCHAR);

INSERT INTO Certificates SELECT DISTINCT certification\_name, content FROM Prak\_Certify;

**GENRE**

-----------------------------

CREATE TABLE Genre(genre VARCHAR PRIMARY KEY);

INSERT INTO Genre SELECT DISTINCT genre From Prak\_Video;

**VIDEO INFO**

-----------------------------

CREATE TABLE VideoInfo(vid INT PRIMARY KEY, title VARCHAR, release\_year INT, producer VARCHAR, color VARCHAR, country VARCHAR, certification\_name VARCHAR REFERENCES Certificates(certification\_name));

INSERT INTO VideoInfo (SELECT DISTINCT v.vid, v.title, v.release\_year, v.producer, v.color, v.country, c.certification\_name FROM Prak\_Video as v, Prak\_Certify as c WHERE v.vid = c.vid);

**BELONG TO GENRE**

----------------------------

CREATE TABLE BelongToGenre(vid INT REFERENCES VideoInfo(vid), genre VARCHAR REFERENCES Genre(genre), PRIMARY KEY (vid, genre));

INSERT INTO BelongToGenre (SELECT DISTINCT vid, genre FROM Prak\_Video);

**MOVIE INFO**

----------------------------

CREATE TABLE MovieInfo(mid INT PRIMARY KEY, mtitle VARCHAR);

INSERT INTO MovieInfo (SELECT DISTINCT v.vid, v.title FROM Prak\_Video AS v WHERE v.vid IN (SELECT DISTINCT tvid FROM Prak\_TVEpisode));

**RATINGS**

--------------------------

CREATE TABLE Ratings(vid INT REFERENCES VideoInfo(vid), uid INT references UserInfo(uid), rate\_score INT, ratingtime TIME, PRIMARY KEY(vid, uid, ratingtime));

INSERT INTO Ratings (SELECT DISTINCT vid, uid, rate\_score, rate\_time FROM Prak\_Users WHERE vid IS NOT NULL);

**FRIEND REQUESTS**

------------------------

CREATE TABLE FriendRequests(uid1 INT REFERENCES UserInfo(uid), uid2 INT REFERENCES UserInfo(uid), requesttime TIME, response VARCHAR, responsetime TIME, PRIMARY KEY(uid1, uid2, requesttime), check(uid1<>uid2));

INSERT INTO FriendRequests SELECT DISTINCT user1, user2, request\_time, response, response\_time FROM Prak\_Friends\_Request;

**FRIENDS**

-------------------------

CREATE TABLE Friends(uid1 INT REFERENCES UserInfo(uid), uid2 INT REFERENCES UserInfo(uid), PRIMARY KEY (uid1, uid2));

INSERT INTO Friends (SELECT uid1, uid2 FROM FriendRequests WHERE response = 'Y');

INSERT INTO Friends (SELECT uid2, uid1 FROM FriendRequests WHERE response = 'Y');

**ACT IN**

--------------------------

CREATE TABLE ActIn(vid INT REFERENCES VideoInfo(vid), pid INT REFERENCES Performer(pid), PRIMARY KEY (vid, pid));

INSERT INTO ActIn (SELECT DISTINCT vid, pid FROM Prak\_Act);

**TV EPISODE INFO**

-------------------------

CREATE TABLE TVEpisodeInfo(tvid INT PRIMARY KEY REFERENCES VideoInfo(vid), episode\_num INT);

INSERT INTO TVEpisodeInfo (SELECT DISTINCT tvid, episodenum FROM Prak\_TVEpisode);

**COLLECTION**

-------------------------

CREATE TABLE Collection(collection\_id INT PRIMARY KEY REFERENCES VideoInfo(vid), collection\_name VARCHAR);

INSERT INTO Collection (SELECT DISTINCT collectionid, collection\_title FROM Prak\_InCollection);

**SEASON**

--------------------------

CREATE TABLE Season(season\_title VARCHAR, season\_num INT, PRIMARY KEY(season\_title, season\_num));

INSERT INTO Season (SELECT DISTINCT seasontitle, seasonnumber FROM Prak\_TVEpisode);

**IN SEASON**

--------------------------

CREATE TABLE InSeason(tvid INT REFERENCES TVEpisode(tvid), season\_title VARCHAR, season\_num INT, FOREIGN KEY (season\_title, season\_num) REFERENCES Season(season\_title, season\_num), PRIMARY KEY(tvid, season\_title, season\_num));

INSERT INTO InSeason (SELECT DISTINCT tvid, seasontitle, seasonnumber FROM Prak\_TVEpisode);

**IN COLLECTION**

---------------------------

CREATE TABLE InCollection(collection\_id INT REFERENCES Collection(collection\_id), season\_title VARCHAR, season\_num INT, PRIMARY KEY(collection\_id, season\_title, season\_num), FOREIGN KEY(season\_title, season\_num) REFERENCES Season(season\_title, season\_num));

INSERT INTO InCollection (SELECT collectionid, seasontitle, seasonnumber FROM Prak\_InCollection);

**MOVIE DIRECTORS**

---------------------------

CREATE TABLE MovieDirectors(did INT REFERENCES Director(did), vid INT REFERENCES VideoInfo(vid), PRIMARY KEY (did, vid));

INSERT INTO MovieDirectors (SELECT DISTINCT did, vid FROM Prak\_VideosDirect WHERE vid IN (SELECT DISTINCT mid FROM MovieInfo));

**TV EPISODE DIRECTORS**

---------------------------

CREATE TABLE TVEpisodeDirectors(did INT REFERENCES Director(did), tvid INT REFERENCES TVEpisodeInfo(tvid), PRIMARY KEY (did, tvid));

INSERT INTO TVEpisodeDirectors (SELECT DISTINCT did, vid FROM Prak\_VideosDirect WHERE vid IN (SELECT DISTINCT tvid FROM Prak\_TVEpisodes));

**Querying the Database**

We used the following five queries on our webpage:

1. List 10 latest videos that the actor “Brad Pitt” participated.

SELECT title

FROM VideoInfo

WHERE vid IN (SELECT vid FROM ActIn WHERE pid = (SELECT pid FROM

Performer WHERE first\_name = 'Brad' AND last\_name =

'Pitt')) ORDER BY release\_year DESC LIMIT 10

1. List the user who has the most friends.

SELECT first\_name, last\_name

FROM UserInfo, (SELECT uid1 AS u, COUNT(\*) AS c FROM Friends GROUP BY

uid1 ORDER BY c DESC LIMIT 1) AS f

WHERE f.u = uid

1. List the performer who has been in the most movies.

SELECT first\_name, last\_name

FROM Performer, (SELECT ActIn.pid, COUNT(\*) as c FROM ActIn WHERE vid IN

(SELECT mid FROM MovieInfo) GROUP BY ActIn.pid ORDER

BY c DESC LIMIT 1) AS P

WHERE Performer.pid = P.pid

1. List 10 videos in the Fantasy genre.

SELECT title

FROM VideoInfo

WHERE vid IN (SELECT vid FROM BelongToGenre WHERE genre = 'Fantasy') LIMIT 10

1. List all friends of John Abbey

SELECT first\_name, last\_name

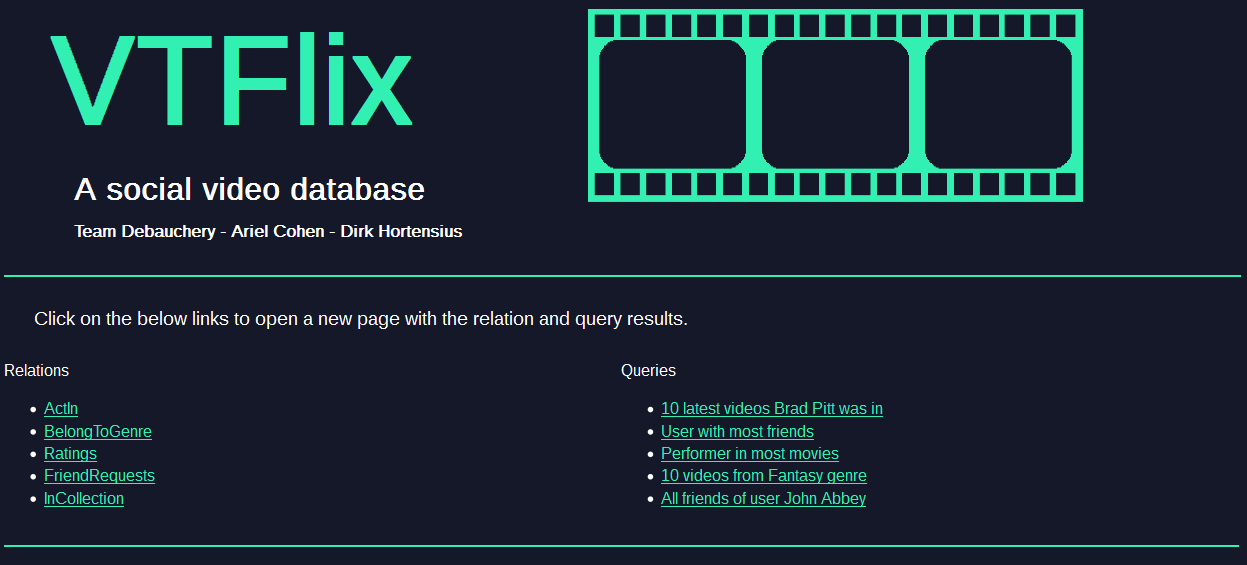
FROM UserInfo

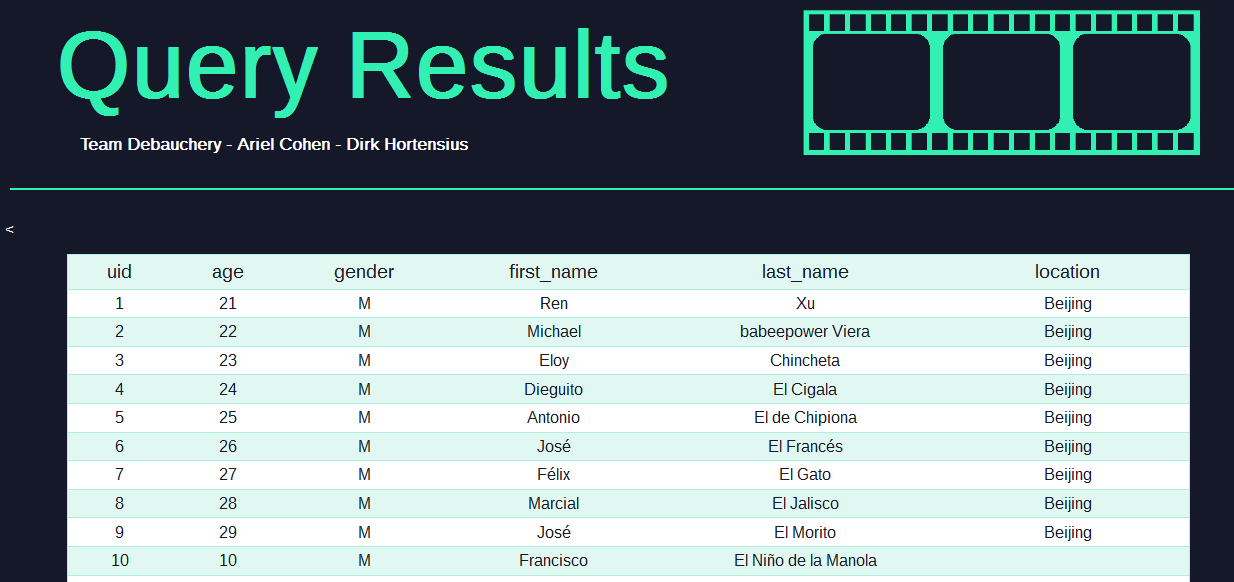
WHERE uid IN (SELECT uid FROM Friends WHERE uid1 = (SELECT uid FROM

UserInfo WHERE first\_name = 'John' AND last\_name = 'Abbey'))

**Extra Functionality**

**Sophisticated Interface**: Below is a snap shot of the top of our interface and an example output.





**Movie Recommendations**: Given a uid, we can output the titles of movies that your

friends have given more than three stars to.

**Safe Queries:** Insures that only “select” commands are allowed in the adhoc query

line so users cannot delete our database.