CS-410/CS-510: High-stakes Writing Assignment
Data-intensive Application Development:
Movie Tracking and Theater Management

Nicholas Alexander
Daniel Davis
Katarina Richards

May 3, 2015

Contents

1 Data-intensive Application Selection

Describe the data-intensive application you have selected. What are its characteristics? What makes it data-intensive? Who is the sponsor of this project? Who are the end-users? How will they benefit from this application?

For our data-intensive application we chose to have an application that works as a movie theatre showtime website, which we feel would be highly supported by other movie theatre chains due to it catering to large demographics. This movie theatre not only shows new movies, but it also shows movies throughout the centuries. In order to facilitate ease of use for users, this web application will enable users to look up information about each movie including the actors, actresses,the director(s), producers, and even the year it was released. The functionality extends beyond that by also allowing people to search for movies by date, showtimes, ticket prices, theatre locations, movie title, and genre. Users can purchase tickets for a movie at a specific showtime and location. Moreover, if a movie is missing from the database, an employee will have the option to add a movie into the database as well as any other information related to it, which could potentially be chosen for a showing at theatres. Our end-users would constitute the employees of the theatres and the people who wish to see specific movies.

The idea is that a movie theatre would be able to gather not only casual movie watchers but also movie critics and film buffs for the newest movie releases as well as for showings of older films on the big screen, where cost of attendance would be at a much cheaper price. As such, this requires that we have a significant range of movies that cater to many different audience types. Therefore, our data would be pulled from the Internet Movie Database(IMDB) which has the aforementioned tables(movies, genres, actors/actresses, producers, directors) with an extremely large amount of tuples. These tables provide us with a massive amount of data which employees and movie-goers can query for general information related to the movie. Additional tables that we would add to make the database schema more complete would be theatre and showtimes, enabling these people to easily find movie showtimes near their locations. Furthermore, our database web application would also be useful for theatre managers in that they could easily find movies from specific time periods, add showtimes for movies, and even add new theatre locations. With this capability, each theatre could even host special events where they show some iconic films from certain eras, which would draw from different demographics every time the era changes. An example of this would be a movie theatre showing movies that were released during the 1950s for a few weeks, and then showing movies from the 1970s for another period of time. Thus, since we are manipulating and querying thousands of tuples of data, this would constitute a data intensive application.

There are tremendous benefits from using this web application. This movie showtime application for theatre chains allows users to not only search for specific showtimes and locations, but it also enables users to search for movies with certain actors/actresses, movies that were released during a specific era, and even search for genres that suit their mood. Furthermore, users can look for movies that are shown at a specific time or range of times in order to find a list of movies that are being shown that will fit within their hectic schedules. In the event that a movie-lover can't find a movie they desire within the database, they can easily and efficiently add it as well as any pertinant information. Furthermore, as stated before, employees can easily set movie showtimes, locations, and ticket prices. Overall, this database schema provides a monumental amount of functionality for almost any need for the

both the business and the end user.

2 Identification and Documentation of Use-cases

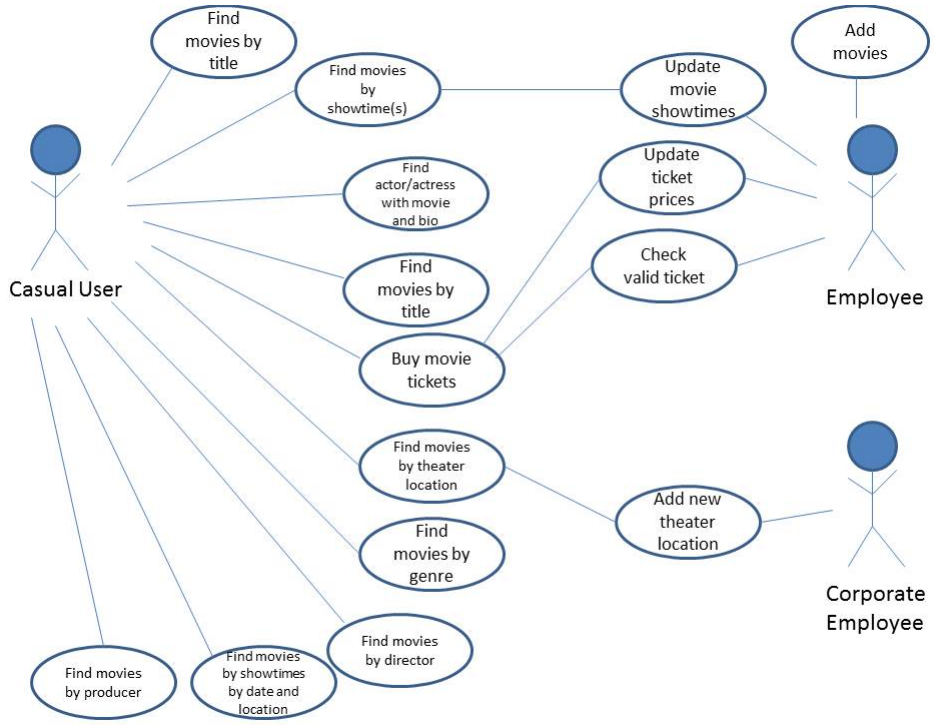
How may user classes do you have? How many actors (including human, application, and abstract ones like time)? Name use-cases. Document them using the LaTeX template.

Since we did not implement an actual program due to time constraints, we have no way of knowing how many classes would have been created for this particular implementation or what those specific classes would be. However, we are certain that the main actors that would affect this database schema would be the casual users doing various transactions and querying the database for information on showtimes for movies as well as looking up the general information for movies. The other actors that will affect the schema are the employees who will be setting the showtimes and ticket prices. The table below illustrates the use cases.

Case #	Actor(s)	Scenario	Outcomes	Failure Condition(s)
1	Casual user	Queries for a movie by title	Movie(s) with that name are displayed in the view	The movie doesnt exist. Display an error message stating the movie doesnt exist.
2	Casual user	Queries for a specific showtime or showtime range	Movie(s) shown during the specified showtime(s) are displayed to the user	There are no movies showing at that time. Display an error message saying that there are no movies during those times.
3	Casual user	Queries for a specific actor/actress	Information pertaining to that actor/actress is returned including their biography and movies they have starred in	The actor/actress is not in the database. Display an error message stating that the actor/actress could not be found.
4	Casual user/Employee	Queries for movies from a particular time period	Movie(s) released during that time period are displayed	No movies were released during the specified time period. Display an error message to the user.
5	Casual user	Queries for movies in a particular theatre location	All movie(s) and showtime(s) are displayed to the user at the specified location	No movies are being shown at that location. Display an error to the user.
6	Casual user	Queries for a movies by genre	All movie(s) that belong to that genre are displayed to the user	No movies belong to the specified genre. Display an error message.
7	Casual user	Queries for movies by director	All movie(s) directed by specified diretor displayed	The specified director hasn't directed any films. Display the error to the user.
8	Casual user	Queries for showtimes by a specified date and location	All movie(s) and showtime(s) for that location and date are shown to the user.	There are no movies beng shown on that date and at that location. Display an error to the user.
9	Casual user	Queries for movies based on producer(s)	All movie(s) produced by the specified producer(s) are displayed to the user.	The specefied producer(s) have not produced any films. Display error.
10	Employee	Chooses to add a movie by providing the necessary info (i.e the year, movie title, genre, actor/actress, etc)	The film is added to the database.	The film already exists or they did not provide the necessary information. Display an error message.

11	Employee	Wishes to update movie showtimes for a specific movie at a specific theatre location	The movie is updated with the specified showtimes.	The showtime already exists, the movie doesn't exist, or the movie isn't showing at that location. Display an error message to the employee.
12	Corporate Employee	Wishes to add a new theatre location	theatre table updated with new location and ID.	The theatre already exists. Display an error
13	Employee	Wishes to update the prices for a particular movie at a specific theatre location	The movie is updated with the new ticket price.	The ticket price is a negative or invalid value. Display an error message to the user.
14	Casual User	Wishes to buy a ticket to a movie	The user obtains a printable version of their ticket and is able to go see the movie at the specified showtime/location. Ticket is considered used and cannot be reused.	Invalid, forged, or already used tickets will not be accepted. They cannot go to the movie.

3 Use-case Diagram



4 Identification and Documentation of Data Tasks in the Application

Use case 1: A casual users inputs the title of a movie they wish to search for, which would be a query on the Movies table. Information regarding the movie is displayed to the user including the genre, a picture of the movie cover, the release date, and the title. The user can then choose to click the link to find more information about the movie such as actors, actresses, directors, producers, and locations that are showing this movie as well as showtimes. If the movie doesn't exist, an error will occur and the view will return a page that displays an error that says that no search results were found.

Use case 2: A casual user inputs the showtimes or range of showtimes that fit their schedule, which would be a query on the Showtimes table. This would then result in the user seeing a view that displays the showtimes for all movies that have showtimes as well as the locations of theaters that are showing them. If there are no movies that are being shown during the time(s), an error would occur, and the user would see a view that displays a message basically stating that there are no movies showing at that time.

Use case 3: A casual user inputs the name of an actor or actress, which woudl be a query on the Actor table. The resulting page would be a list of actor(s) with the specified name. From there, the user could choose to click a link to the actor/actress of their choice to find out more information about them. If the actor or actress did not exist in the database, no

search results would be found and an error message would be displayed to the user.

Use case 4: A casual user inputs the release date for movies, which would query the Movie table. The results of this query would be all movies that were released during that specific year. However, if no movies were not related during the specified year, an error would occur and an error message would be displayed to the user that states there were no search results found.

Use case 5: A casual user inputs the theater location that they wish to see a movie at, which would query the Theatre, Showtime, and Movie tables. The resulting outputs would be all movies and showtimes at that particular location. If the location did not exist, an error would occur and the user would see a message that says that there are no theatrs at that location.

Use case 6: A casual user inputs a particular movie genre they wish to see, which would be a query on the Movie table. All movies that fit the specified genre would be displayed to the users. If there are no movies that exist for that specific genre, an error would occur, and the user would see a message that states there are no movies for that particular genre.

Use case 7: A casual user inputs a director's name to see what movie's he or she has directed. If it is successful, the output to the user would be a complete list of movies that were directed by the specified director, which would be query on the Director and Movie tables. If it fails, this would be due to that director either not existing or he/she has not directed any movies. Therefore, in the event of failure, an error message would be displayed to the user stating the director has not directed any films.

Use case 8: A casual user inputs showtimes for a specified date and location, which would be a query on the Showtimes, Theatre, and Movie tables. The outputs put for this particular query would be ll movie(s) and showtime(s) for that location and date. In the event that there are no movies showing on that date for that location, the user would see an error message stating that there are no movives showing for that time.

Use case 9: A casual user inputs producer(s)ś names. A list of producers with that name are displayed. If the user clicks on their name, a list of movies they have produced will be displayed, which would be a query on the Producer and Movie tables. If the producer does not exist, an error message will be displayed that states the specefied producer(s) have not produced any films.

Use case 10: An employee wishes to add a movie by providing the necessary info. They would input the year, movie title, genre, roles, and director, which would affect the Movie, director, producer, and role table. The output of this transaction would be adding a movie as well as the information to the tables mentioned previously. The fail conditions for this specific query would be that the film already exists or they did not provide the necessary information. An error message would be displayed to th user, explaining whether it failed due to the film already existing or if it was a failure due to invalid input.

Use case 11: an employee wishes to update movie showtimes for a specific movie at a specific theater location. They would input the movie id, the showroom id, the date it shows, the start time, the end time, and what theater the movie is showing. This transaction would affect the Showtime table. The outputs of this transaction would be that the movie is updated with the specified showtimes. The fail conditions of this use case would be if the showtime already exists, the movie doesn't exist, or the movie isn't showing at that location. In any of these scenarios, an error message must be displalyed to the user to let them know where it failed so that they can correct the mistake.

Use case 12: A corporate employee wishes to add a new theater location. The corporate employee would input the necessary information such as name, city, state, and zip, which would affect the Theatre table. The outputs would be that the theater table updated with the new location and ID. If it fails, it would be because the theater already exists, and an error would be displayed to the user.

Use case 13: An employee wishes to update the prices for a particular movie at a specific theater location. This means the employee would have to input a showtime id, seatnumber and a price. The outputs would be that showing of that movie is updated with the new ticket price, which would affect the Ticket table. This transaction would fail if the ticket price is a negative or invalid value in which cae an error message would be displayed to the user.

Use case 14: A casual User wishes to buy a ticket to a movie. They have to input a showtime and a seat number, which is a transaction on the Ticket table. The output is that the user obtains a printable version of their ticket which has the showing of the movie, the movie title, and the seat number. This will fail if the ticket showtime for that particular movie does not exist or a seat number is already taken, which means a message will be displayed to the user accordingly.

5 Identification and Documentation of Transactions

1. Creating a Theater
   When creating a theater, showrooms are typically also added at the same time. This transaction will not be executed extremely frequently since new theaters are typically not constructed extraordinarily often. However, it will see a modest amount of usage.

2. Add a ticket to a showtime
   When a new showtime opens for a movie, there must be tickets for that showtime. As such, this transaction will get executed very frequently. New movies are constantly releasing and thus new showtimes meaning new tickets.

3. Add a movie
   Similar to adding a ticket to a showtime, new movies are constantly being added to a theater's showings. When a movie is added, the directors and producers must also be added along with possible actors and actresses. Since new movies are constantly being released, this transaction will be executed more than any other transaction.

4. Update ticket prices
   Sometimes a movie theater may find the need to update the ticket prices. While this may not be very often, it does happen. This transaction will most likely not be executed too frequently, but it is substantial enough to be mentioned.

5. Delete all showtimes for a movie
   After a movie has played for a while, it must be removed from the showings. Again, since movies are constantly going in-and-out, this transaction will be executed very frequently - on the same order as adding a movie.

6 Identification and Documentation of Database Queries

Describe your database queries (in English, not in SQL) here.

1. The user wants to view all the movie titles, the actors and actresses that play in that movie, and the characters that the actor or actress plays in the film. This query will be executed fairly frequently to show which actors or actresses star in a film.

2. The user wants to view the directors and producers of each movie. This query will be executed on the same magnitude as the actors and actresses in order to give credit to directors and producers.

3. The user wants to know what all movies were released in a specific year, such as 2015. This query will be executed somewhat normally.

4. The user wants to view the average price for a ticket at a certain theater. This will be executed extremely frequently as consumers always want to find the best deal.

5. The user wants to view all theater locations. This will also be executed extremely frequently since consumers need to know where movie theaters exist.

6. The user wants to view all theaters that are currently playing a specific movie. This will be executed on the same magnitude of the previous two queries since customers want to know what movies are playing.

7. The user wants to know all information regarding the showtime – such as the start time, show date, and ticket price – for all showings of a specific movie. This query is probably the most frequent query since a user must always check this before attending a movie.

8. Similar to the previous query, the user wants to see all the same information for all movies. This query will be executed somewhat less frequently than the previous since customers typically have an idea of what move they want to see before searching (through means such as TV advertising).

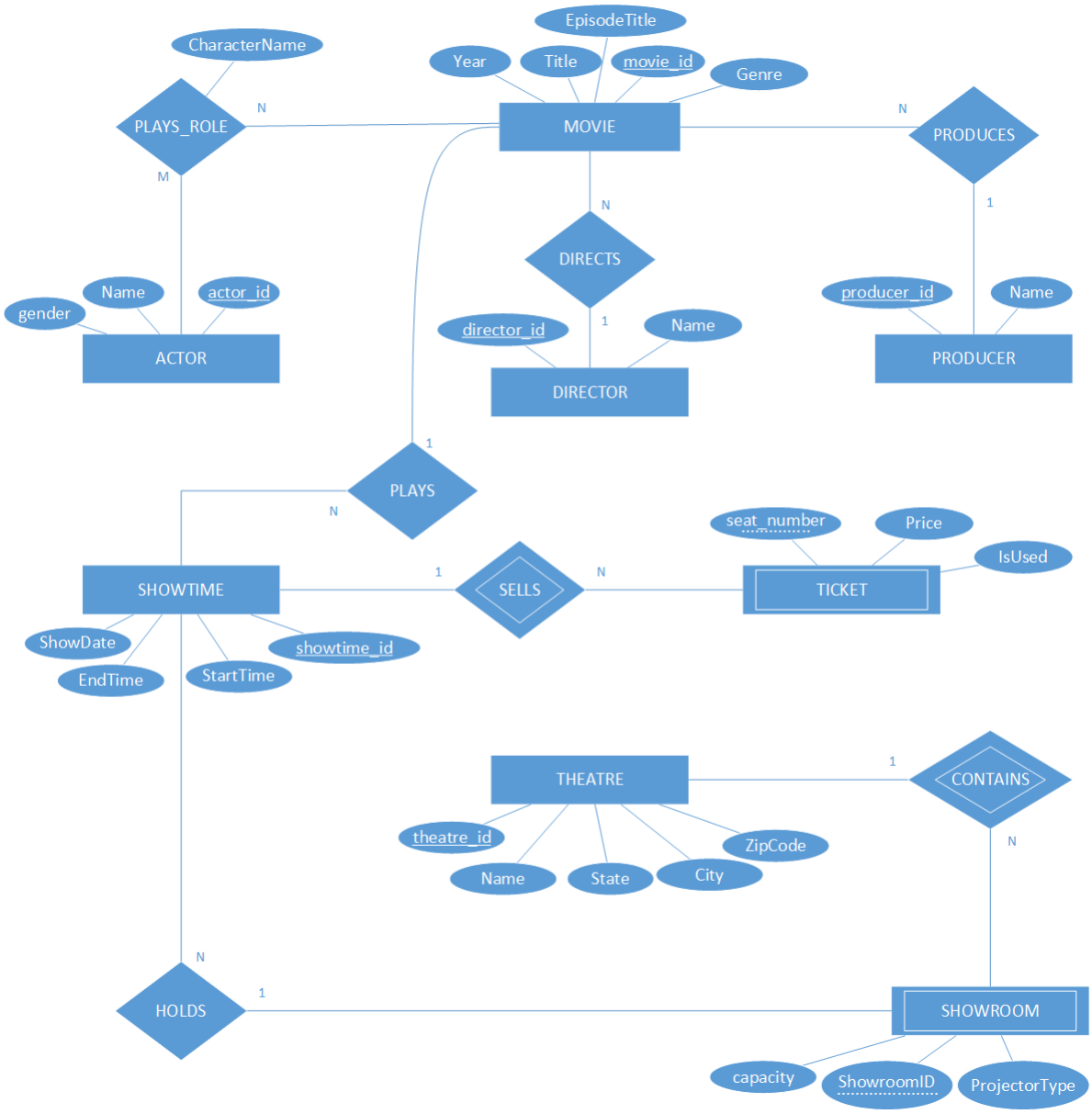
9. The user wants to view all movies that are director and/or produced by a specific producer/director. This query may be executed somewhat infrequently, but it may happen if the user wants to see similar titles based on director/producer.

10. The user wants to see what types of projectors, and how many, a theater uses in its showrooms. This may be executed semi-frequently by a corporate office in order to prioritize what theaters get new projectors and when.

11. The user wants to see the number of tickets sold and redeemed as well as the sum of the sales. This query will likely be the second most-frequently executed query since businesses always keep close tabs on sales.

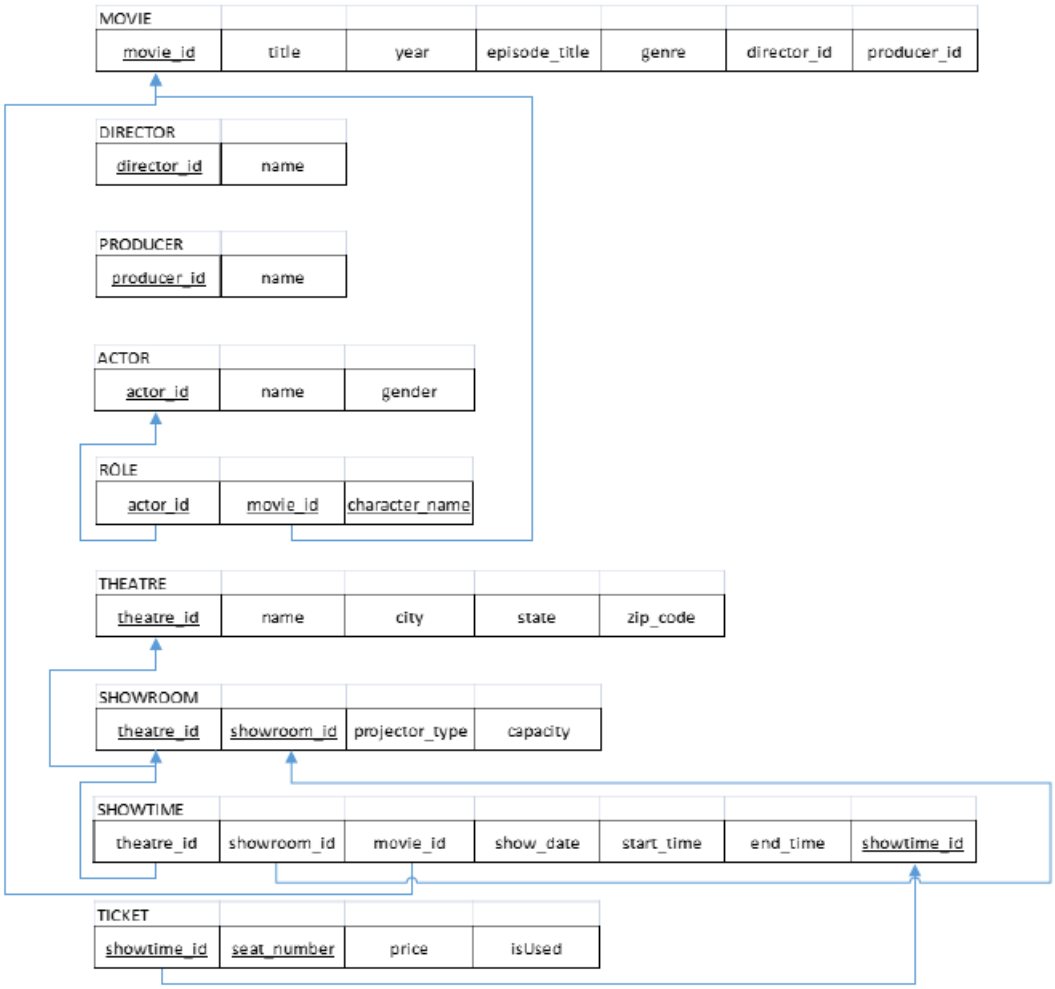
12. This query is almost identical to the previous but allows the user to also see how many tickets are still available for a showtime. This query may be executed sometimes if a user needs to know if there are tickets available for purchase.

7 Conceptual Data Model



8 Logical Data Model

The relational mapping that follows was produced using the standard mapping techniques discussed in lecture. All many-to-many relationship types were handled through the inclusion of a separate relation, and all one-to-many relationship types identified in the conceptual model were mapped by placing pushing the appropriate attributes to the "many" side of the relationship.



We now analyze the functional dependencies determined from the above relational mapping, listed by table. The given list of functional dependencies will then be used in explaining the degree of normality for the schema.

MOVIE:
movie_id → title, year, episode_title, genre, director_id, producer_id

DIRECTOR:
director_id → name

PRODUCER:
producer_id → name

ACTOR:
actor_id → name, gender

ROLE:

The ROLE relation exhibits no functional dependencies. This is because it represents a relationship type, rather than an entity type. Thus, all of its attributes form the primary key (superkey).

THEATRE:
theatre_id → name, city, state, zip_code

SHOWROOM:
theatre_id, showroom_id → projector_type, capacity

SHOWTIME:
showtime_id → theatre_id, showroom_id, movie_id, show_date, start_time, end_time

TICKET:
showtime_id, seat_number → price, isUsed

For a relation to satisfy First Normal Form (1NF), it must be true that each attribute value represents an atomic (indivisible) value. More simply, there must be no composite attributes present in the relation. As can be seen in the logical database model above, no composite attributes exist for any relation in the schema. Thus, the schema satisfies 1NF.

For a relation to satisfy Second Normal Form (2NF), every nonprime attribute of the relation must be fully functionally dependent upon the primary key. That is to say that all left side attributes for each functional dependency are required for the functional dependency to hold. In each case above, all other attributes of each relation are functionally determined by the primary key. This means that each relation exhibits full functional dependency, and thereby the schema satisfies 2NF.

A relation satisfies Third Normal Form (3NF) if it satisfies 2NF and no attributes of the relation exhibit transitive dependency upon the primary key. That is to say that, for a relation R, with primary key, K, no attribute X exists such that, $X \to Y$ and $Y \to K$. It can be seen from the enumeration of functional dependencies given above that this is indeed the case. Each attribute of each relation is only functionally dependent upon the primary key, and thus 3NF is satisfied.

9 Physical Data Model

To analyze the physical data model for this relational schema, we consider each table separately. For each, we examine an appropriate choice of attribute to utilize for sorting, and select an indexing scheme based upon how we anticipate the table would most frequently be queried in the application under design.

MOVIE:
Because we have designed the application to allow customers to search for specific movies, we expect that the table will most commonly be queried by name. However, because new

movies are constantly being released, and thus will need to be added to the database, we have provided an auto-incrementing $movie_id$ as the primary key. This ordering field will allow for efficient insertion, since its auto-incrementing nature will not require sorting upon each insertion. Then, to allow for maximum efficiency in querying, we provide data access through a secondary index on the non-ordering attribute, title.

DIRECTOR:

Similarly to above, we know that directors will most often be searched for by name, but we also need to consider the efficient insertion of new directors. Thus, the primary key and ordering field for the relation is provided by an auto-incrementing director_id attribute, while data access will occur through a secondary indexing scheme on the name attribute.

PRODUCER:

The producer relation follows the exact same reasoning as that of the DIRECTOR relation. We must be able to efficiently insert new directors while maintaining the ability to efficiently query by name, as that will be the most common query. Therefore, we provide the auto-incrementing integer, producer_id as the primary key for the relation, while forming a secondary index on the name attribute.

ACTOR:

The only physical difference between the ACTOR relation and the two previous relations is the addition of the gender attribute (indicating whether the individual is an actor or actress, included due to the separation of the two in the sample input data). This difference does not change how the data should be physically stored, as it is not likely that the table will frequently be queried for all members of either category. Thus, the need for efficient insertion along with efficient querying by name remain the two primary goals. As before, they are met by utilizing the auto-incrementing primary key, actor_id, as the ordering field, while providing data access through a secondary indexing scheme on the name attribute.

ROLE:

It should be noted that the ROLE relation is not an entity type, but rather a relationship type, representing a relationship between the ACTOR and MOVIE relations. As such, it is not likely to be queried directly, but rather used as part of a JOIN operation which includes the two entity types it relates. It should also be noted that the primary key for this relation is a superkey consisting of the three attributes actor_id, movie_id, and character_name. Thus, as discussed in lecture, all three components will be used in ordering the stored data. Thus, the tiered structure for ordering was selected to be actor_id¿¿movie_id¿¿charcter_name, as this logically groups the tuples by actor_id. It is impossible to determine without the application entering production whether it would be more common to query which movies an actor was in, or what actors played a role in a given movie. Thus, the most efficient choice for indexing cannot be made with certainty. Either case warrants a clustering index scheme, since, while both the actor_id and movie_id attributes are ordering fields, neither are unique on their own.

THEATRE:

In the case of the THEATRE tuple, the primary key was once again selected to be an autoincrementing theatre_id. However, unlike in previous relations, this selection was not made with efficiency in mind. It is possible that two theatres with the same name may exist

in the same city. (This may occur when a chain of theatres operating under the same name open multiple locations in a large metropolitan area.) Thus, even the combination of all four other attributes would not satisfy the uniqueness constraint required to define a primary key. Thus, the ordering for the table obviously occurs on the theatre_id attribute. However, it is not likely that the table will frequently be queried by the primary key. Thus, a secondary index on the zip code was chosen as the indexing scheme, since it will likely be very common for users to query the system by zip code to locate theatres near their location.

SHOWROOM:

The SHOWROOM relation is a weak entity type which has an identifying relationship with the THEATRE entity type. As such, the primary key is a superkey, consisting of the theatre_id of the containing theatre, and the showroom_id of the room in that theatre. Logically then, the ordering for this table is performed first by theatre_id, and second by showroom_id. This table will not experience a comparatively high volume of insertions, as new theatres open far less frequently than movie releases or new actors appearing. Querying the table will likely occur in the situation of looking for all showrooms in a specific theatre, so a primary indexing scheme on the theatre_id attribute would be most efficient.

SHOWTIME:

It was originally intended to use a superkey consisting of the theatre_id, showroom_id, show_date, and start_time attributes as the primary key for the relation. However, the DBMS reported that the proposed superkey could not satisfy the uniqueness constraint. Logically, it does, as only one movie can be played in a single showroom, at a single theatre, at a single time, on a single day. However, none of these attributes are unique individually, and we hypothesize that is the reason that PostgreSQL would not accept the initially chosen primary key. Thus, we chose to incorporate the auto-incrementing showtime_id as the primary key. However, this means that the ordering field, showtime_id, is nearly useless for querying the database. It is far more likely that users will query showtimes for either a specific theatre or specific day. Since the application is not in production, we cannot know which query would be more frequent in reality, but we believe that searching by theatre would occur more frequently. Thus, we selected to use a secondary indexing scheme on the theatre_id attribute.

TICKET:

Finally, the TICKET relation is used to store purchased tickets to a given showtime. Thus, it is most likely that it will be most frequently queried by the showtime_id attribute. Showtime_id and seat_number already form the primary (super)key, for the relation, so the table is ordered, first by showtime_id, and second by the seat_number. Data access is most efficiently provided through a primary index scheme on the showtime_id attribute.

It should be noted that while we have discussed each of these indexing scheme as if a single level index is to be utilized. However, in reality these schemes are implemented as multi-level indeces using B+ Trees, as discussed class lectures. This, however, does not change the appropriateness of the choices of which attributes to use in the index.

10 Database Creation and Data Loading

10.1 Creation

```
1  /*
2      SQL Script to create the Theater database.
3      Domains:
4          - GENDER
5          - PROJECTOR
6
7      Tables:
8          - DIRECTOR
9          - PRODUCER
10         - MOVIE
11         - ACTOR
12         - ROLE
13         - THEATER
14         - SHOWROOM
15         - SHOWTIME
16         - TICKET
17 */
18
19
20 /*
21     Encapsulate binary gender values M for Males and F for Female
22 */
23 CREATE DOMAIN GENDER CHAR(1)
24     CHECK (
25         value IN (
26             'M', 'F'
27         )
28     );
29
30
31 /*
32     Encapsulates values for movie projector types
33 */
34 CREATE DOMAIN PROJECTOR VARCHAR(9)
35     CHECK (
36         value IN (
37             'FILM_ROLL', 'DIGITAL', 'LASER', 'IMAX'
38         )
39     );
40
41
42 /*
43     Represents a movie director
44         - director_id    Unique, incremental identifier for a director
45         - name           Name of the director
46 */
47 CREATE TABLE director (
48     director_id      SERIAL PRIMARY KEY,
49     name             VARCHAR(255) NOT NULL
50 );
51
52
```

```
53 /*
54     Represents a movie producer
55         - producer_id    Unique, incremental identifier for a director
56         - name           Name of the producer
57 */
58 CREATE TABLE producer (
59     producer_id      SERIAL PRIMARY KEY,
60     name             VARCHAR(255) NOT NULL
61 );
62
63
64 /*
65     Represents a movie
66         - movie_id           Unique, incremental identifier for a movie
67         - title              Title of the film
68         - episode_title      Name of an episode in the case of an Episodic or TV
               Show
69         - year               Year the film was released
70         - genre              Genre of the film
71         - director_id        Identifier of the film director(s) (Foreign Key)
72         - producer_id        Identifier of the film producer(s) (Foreign Key)
73 */
74 CREATE TABLE movie (
75     movie_id         SERIAL PRIMARY KEY,
76     title            VARCHAR(500) NOT NULL,
77     episode_title    VARCHAR(500),
78     year             INT NOT NULL,
79     genre            VARCHAR(255),
80     director_id      INT REFERENCES director(director_id),
81     producer_id      INT REFERENCES producer(producer_id)
82 );
83
84
85 /*
86     Represents an actor/actress in a movie
87         - actor_id       Unique, incremental identifier for an actor/actress
88         - name           Name of the actor/actress
89         - gender         Gender of the actor (male) or actress (female)
90 */
91 CREATE TABLE actor (
92     actor_id             SERIAL PRIMARY KEY,
93     name                 VARCHAR(255) NOT NULL,
94     gender               GENDER
95 );
96
97
98 /*
99     Represents an actor/actress' role in a movie
100        - actor_id           Identifier for an actor/actress (Foreign Key)
101        - movie_id           Identifier for a movie (Foreign Key)
102        - character_name     Name of the character the actor/actress played
103 */
104 CREATE TABLE role (
105     actor_id             INT REFERENCES actor(actor_id),
106     movie_id             INT REFERENCES movie(movie_id),
107     character_name       VARCHAR(255),
```

```
108     PRIMARY KEY          (actor_id, movie_id, character_name)
109 );
110
111
112 /*
113     Represents a movie Theater
114         - theater_id     Unique, incremental identifier for a Theater
115         - name           Name of the Theater
116         - city           City in which the Theater exists
117         - state          State in which the Theater exists
118         - zipcode        Zip code in which the Theater exists
119 */
120 CREATE TABLE theater (
121     theater_id       SERIAL PRIMARY KEY,
122     name             VARCHAR(500) NOT NULL,
123     city             VARCHAR(255),
124     state            CHAR(2),
125     zipcode          CHAR(5)
126 );
127
128
129 /*
130     Represents a showroom in a Theater
131         - showroom_id        Unique identifier for a showroom
132         - theater_id         Identifier of the Theater containing the showroom (
              Foreign Key)
133         - projector_type     Type of projector that the showroom uses
134         - capacity           Number of people the showroom can hold
135 */
136 CREATE TABLE showroom (
137     showroom_id      INT UNIQUE,
138     theater_id       INT REFERENCES Theater(theater_id),
139     projector_type   PROJECTOR,
140     capacity         INT,
141     PRIMARY KEY      (showroom_id, theater_id)
142 );
143
144
145 /*
146     Represents a showtime for a film
147         - showtime_id    Unique, incremental identifier for a showtime
148         - theater_id     Identifier of the Theater playing the showing (Foreign
              Key)
149         - showroom_id    Identifier of the showroom hosting the showing (Foreign
              Key)
150         - movie_id       Identifier of the movie being shown (Foreign Key)
151         - start_time     Time in which the showing begins
152         - end_time       Time in which the showing ends
153         - show_date      Date on which the showing plays
154 */
155 CREATE TABLE showtime (
156     showtime_id      SERIAL PRIMARY KEY,
157     theater_id       INT REFERENCES Theater(theater_id),
158     showroom_id      INT REFERENCES showroom(showroom_id),
159     movie_id         INT REFERENCES movie(movie_id),
160     start_time       TIME,
```

```
161     end_time        TIME,
162     show_date       DATE
163 );
164
165
166 /*
167     Represents a ticket for a showtime
168         - showtime_id   Identifier of the showtime for which the ticket can be
                used (Foreign Key)
169         - seat_number   Seat for the ticket-holder to occupy
170         - price         Price of the ticket
171         - time_bought   Time in which the ticket was purchased
172         - date_bought   Date on which the ticket was purchased
173         - wasUsed       True if the ticket was redeemed
174 */
175 CREATE TABLE ticket (
176     showtime_id     INT REFERENCES showtime(showtime_id),
177     seat_number     SERIAL UNIQUE,
178     price           NUMERIC(1000, 2) NOT NULL,
179     time_bought     TIME,
180     date_bought     DATE,
181     wasUsed         BOOLEAN,
182     PRIMARY KEY     (showtime_id, seat_number)
183 );
```

10.2 Views

```
1  /*
2      This view automatically stores the locations of all Cinemark theaters.
3  */
4  CREATE TEMP VIEW cinemark_location AS
5  SELECT name, city, state, zipcode
6  FROM theater
7  WHERE name LIKE '%Cinemark%';
8
9
10 /*
11     The view stores all show-times for Cinemark theaters including the movie
           title, show date, start time, and ticket price, and is ordered based on
           movie title.
12 */
13 CREATE TEMP VIEW cinemark_showtime_by_movie AS
14 SELECT
15     name, title, start_time, show_date, price
16 FROM
17     theater, movie, showtime, ticket
18 WHERE
19     theater.name LIKE '%Cinemark%'           AND
20     theater.theater_id=showtime.theater_id   AND
21     showtime.movie_id=movie.movie_id         AND
22     ticket.showtime_id=showtime.showtime_id
23 ORDER BY
24     movie.title ASC,
25     theater.theater_id ASC,
26     showtime.show_date ASC,
27     showtime.start_time ASC;
```

```
28
29
30 /*
31      The view is identical to the previous with the exception that tuples are
            ordered by theater name.
32 */
33 CREATE TEMP VIEW cinemark_showtime_by_theater AS
34 SELECT
35      name, title, start_time, show_date, price
36 FROM
37      theater, movie, showtime, ticket
38 WHERE
39      theater.name LIKE '%Cinemark%'              AND
40      theater.theater_id=showtime.theater_id  AND
41      showtime.movie_id=movie.movie_id            AND
42      ticket.showtime_id=showtime.showtime_id
43 ORDER BY
44      theater.theater_id ASC,
45      movie.title ASC,
46      showtime.show_date ASC,
47      showtime.start_time ASC;
```

10.3 Sample Insertion

```
1  /*
2          SQL script to insert sample test data into the Theater database.
3  */
4
5
6  /*
7          Insert Directors into DIRECTOR table.
8          (director_id, name)
9  */
10 INSERT INTO director VALUES (DEFAULT, 'Joss Whedon');
11 INSERT INTO director VALUES (DEFAULT, 'Zack Snyder');
12 INSERT INTO director VALUES (DEFAULT, 'Colin Trevorrow');
13 INSERT INTO director VALUES (DEFAULT, 'Steven Spielberg');
14
15
16 /*
17         Insert Producers into PRODUCER table.
18         (producer_id, name)
19 */
20 INSERT INTO producer VALUES (DEFAULT, 'Victoria Alonso');
21 INSERT INTO producer VALUES (DEFAULT, 'Wesley Coller');
22 INSERT INTO producer VALUES (DEFAULT, 'Steven Spielberg');
23 INSERT INTO producer VALUES (DEFAULT, 'Kathleen Kennedy');
24
25
26 /*
27         Insert movies into MOVIE table.
28         (movie_id, title, episode_title, year, genre, director_id, producer_id)
29 */
30 INSERT INTO movie VALUES (DEFAULT, 'Avengers: Age of Ultron', NULL, 2015, '
       Action', 1, 1);
31 INSERT INTO movie VALUES (DEFAULT, 'Batman v Superman: Dawn of Justice', NULL,
```

```
    2016, 'Action', 2, 2);
32 INSERT INTO movie VALUES (DEFAULT, 'Jurassic World', NULL, 2015, 'Action', 3,
    3);
33 INSERT INTO movie VALUES (DEFAULT, 'The Lost World: Jurassic Park', NULL, 1997,
     'Action', 4, 4);
34
35
36 /*
37         Insert actor into ACTOR table.
38         (actor_id, name, gender)
39 */
40 INSERT INTO actor VALUES (DEFAULT, 'Robert Downey Jr.', 'M');
41 INSERT INTO actor VALUES (DEFAULT, 'Chris Hemsworth', 'M');
42 INSERT INTO actor VALUES (DEFAULT, 'Mark Ruffalo', 'M');
43 INSERT INTO actor VALUES (DEFAULT, 'Chris Evans', 'M');
44 INSERT INTO actor VALUES (DEFAULT, 'Scarlett Johansson', 'F');
45 INSERT INTO actor VALUES (DEFAULT, 'Jeremy Renner', 'M');
46
47 INSERT INTO actor VALUES (DEFAULT, 'Gal Gadot', 'F');
48 INSERT INTO actor VALUES (DEFAULT, 'Jason Mamoa', 'M');
49 INSERT INTO actor VALUES (DEFAULT, 'Henry Cavill', 'M');
50 INSERT INTO actor VALUES (DEFAULT, 'Ben Affleck', 'M');
51 INSERT INTO actor VALUES (DEFAULT, 'Lois Lane', 'F');
52 INSERT INTO actor VALUES (DEFAULT, 'Ezra Miller', 'M');
53 INSERT INTO actor VALUES (DEFAULT, 'Jesse Eisenburg', 'M');
54
55 INSERT INTO actor VALUES (DEFAULT, 'Vincent DOnofrio', 'M');
56
57 INSERT INTO actor VALUES (DEFAULT, 'Jeff Goldblum', 'M');
58 INSERT INTO actor VALUES (DEFAULT, 'Julianne Moore', 'F');
59
60
61 /*
62         Insert role into ROLE table.
63         (actor_id, movie_id, character_name)
64 */
65 INSERT INTO role VALUES (1, 1, 'Tony Stark / Iron Man');
66 INSERT INTO role VALUES (2, 1, 'Thor');
67 INSERT INTO role VALUES (3, 1, 'Bruce Banner / Hulk');
68 INSERT INTO role VALUES (4, 1, 'Steve Roges / Captain America');
69 INSERT INTO role VALUES (5, 1, 'Natasha Romanoff / Black Widow');
70 INSERT INTO role VALUES (6, 1, 'Clint Barton / Hawkeye');
71
72 INSERT INTO role VALUES (7, 2, 'Diana Prince / Wonder Woman');
73 INSERT INTO role VALUES (8, 2, 'Arthur Curry / Aquaman');
74 INSERT INTO role VALUES (9, 2, 'Clark Kent / Superman');
75 INSERT INTO role VALUES (10, 2, 'Bruce Wayne / Batman');
76 INSERT INTO role VALUES (11, 2, 'Lois Lane');
77 INSERT INTO role VALUES (12, 2, 'Barry Allen / The Flash');
78 INSERT INTO role VALUES (13, 2, 'Lex Luthor');
79
80 INSERT INTO role VALUES (14, 3, 'Vic Hoskins');
81
82 INSERT INTO role VALUES (15, 4, 'Ian Malcom');
83 INSERT INTO role VALUES (16, 4, 'Sarah Harding');
84
```

```
85
86 /*
87         Insert theater into THEATER table.
88         (theater_id, name, city, state, zip-code)
89 */
90 INSERT INTO theater VALUES (DEFAULT, 'Cinemark: Huntington Mall', '
       Barboursville', 'WV', '25504');
91 INSERT INTO theater VALUES (DEFAULT, 'Cinemark: Cinema 10', 'Ashland', 'KY', '
       41101');
92 INSERT INTO theater VALUES (DEFAULT, 'Marquee Cinemas: Pullman Square', '
       Huntington', 'WV', '25701');
93
94
95 /*
96         Insert showroom into SHOWROOM table.
97         (showroom_id, theater_id, projector_type, capacity)
98 */
99 INSERT INTO showroom VALUES (1, 1, 'LASER', 75);
100 INSERT INTO showroom VALUES (2, 1, 'LASER', 75);
101 INSERT INTO showroom VALUES (3, 1, 'LASER', 75);
102 INSERT INTO showroom VALUES (4, 1, 'IMAX', 100);
103 INSERT INTO showroom VALUES (5, 1, 'DIGITAL', 50);
104
105 INSERT INTO showroom VALUES (6, 2, 'IMAX', 125);
106 INSERT INTO showroom VALUES (7, 2, 'DIGITAL', 75);
107 INSERT INTO showroom VALUES (8, 2, 'DIGITAL', 75);
108 INSERT INTO showroom VALUES (9, 2, 'DIGITAL', 75);
109 INSERT INTO showroom VALUES (10, 2, 'DIGITAL', 75);
110 INSERT INTO showroom VALUES (11, 2, 'DIGITAL', 75);
111
112 INSERT INTO showroom VALUES (12, 3, 'LASER', 50);
113 INSERT INTO showroom VALUES (13, 3, 'LASER', 50);
114 INSERT INTO showroom VALUES (14, 3, 'IMAX', 60);
115 INSERT INTO showroom VALUES (15, 3, 'FILM ROLL', 25);
116
117
118 /*
119         Insert show-time into SHOWTIME table.
120         (showtime_id, theater_id, showroom_id, movie_id, start_time, end_time,
              show_date)
121 */
122 INSERT INTO showtime VALUES (DEFAULT, 1, 1, 1, '9:40pm', '12:01am', '5/1/2015')
       ;
123 INSERT INTO showtime VALUES (DEFAULT, 1, 5, 1, '10:00pm', '12:21am', '5/1/2015'
       );
124 INSERT INTO showtime VALUES (DEFAULT, 1, 4, 1, '8:00pm', '10:21pm', '5/1/2015')
       ;
125
126 INSERT INTO showtime VALUES (DEFAULT, 2, 6, 1, '7:00pm', '9:21pm', '5/1/2015');
127 INSERT INTO showtime VALUES (DEFAULT, 2, 11, 1, '8:10pm', '10:31pm', '5/1/2015'
       );
128
129 INSERT INTO showtime VALUES (DEFAULT, 3, 3, 1, '7:10pm', '9:31pm', '5/1/2015');
130
131 INSERT INTO showtime VALUES (DEFAULT, 1, 4, 2, '8:00pm', '10:00pm', '5/1/2016')
       ;
```

```
132 INSERT INTO showtime VALUES (DEFAULT, 2, 6, 2, '8:00pm', '10:00pm', '5/1/2016')
    ;
133 INSERT INTO showtime VALUES (DEFAULT, 3, 14, 2, '8:00pm', '10:00pm', '5/1/2016'
    );
134
135
136 /*
137         Insert ticket into TICKET table.
138         (showtime_id, seat_number, price, time_bought, date_bought, wasUsed)
139 */
140 INSERT INTO ticket VALUES (1, DEFAULT, 7.50, NULL, NULL, FALSE);
141 INSERT INTO ticket VALUES (1, DEFAULT, 7.50, NULL, NULL, FALSE);
142 INSERT INTO ticket VALUES (1, DEFAULT, 7.50, NULL, NULL, FALSE);
143 INSERT INTO ticket VALUES (1, DEFAULT, 7.50, '6:25pm', '4/30/2015', TRUE);
144
145 INSERT INTO ticket VALUES (2, DEFAULT, 7.50, NULL, NULl, FALSE);
146
147 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '6:25pm', '4/30/2015', TRUE);
148 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '6:25pm', '4/30/2015', TRUE);
149 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '6:25pm', '4/30/2015', TRUE);
150 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '11:20am', '5/1/2015', FALSE);
151 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '11:21am', '5/1/2015', FALSE);
152 INSERT INTO ticket VALUES (3, DEFAULT, 10.00, '12:00pm', '5/1/2015', FALSE);
153
154 INSERT INTO ticket VALUES (4, DEFAULT, 7.50, '6:59pm', '5/1/2015', TRUE);
155
156 INSERT INTO ticket VALUES (7, DEFAULT, 12.50, NULL, NULL, FALSE);
157 INSERT INTO ticket VALUES (8, DEFAULT, 12.50, NULL, NULL, FALSE);
158 INSERT INTO ticket VALUES (9, DEFAULT, 12.50, NULL, NULL, FALSE);
```

11 Implementing Database Transactions and Queries

11.1 Sample Query

```
1  /*
2      Query to show all movie titles along with the actors/actresses
3      that perform in the movie as well as the character name.
4  */
5  SELECT DISTINCT
6      title           AS "Movie",
7      name            AS "Starring",
8      character_name  AS "As"
9  FROM
10     movie, actor, role
11 WHERE
12     role.movie_id=movie.movie_id        AND
13     role.actor_id=actor.actor_id
14 ORDER BY
15     title,
16     name;
17
18
19 /*
20     Query to view the directors and producers of all
21     movies.
22 */
```

```
23 SELECT DISTINCT
24     title            AS "Movie",
25     director.name    AS "Director",
26     producer.name    AS "Producer"
27 FROM
28     movie, director, producer
29 WHERE
30     movie.director_id=director.director_id  AND
31     movie.producer_id=producer.producer_id
32 ORDER BY
33     title;
34
35
36 /*
37     Query to view all movie titles that were released before
38     the year 2015
39 */
40 SELECT
41     title AS "Title"
42 FROM
43     movie
44 WHERE
45     year=2015;
46
47
48 /*
49     Query to view the average ticket price for each theater
50 */
51 SELECT
52     theater.name             AS "Theater",
53     round(AVG(price), 2)     AS "Average Ticket Price ($)"
54 FROM
55     showtime, ticket, theater
56 WHERE
57     showtime.theater_id=theater.theater_id  AND
58     ticket.showtime_id=showtime.showtime_id
59 GROUP BY
60     theater.name;
61
62
63 /*
64     Query to view all theaters and their city, state, and zipcode
65     location.
66 */
67 SELECT
68     name,
69     city,
70     state,
71     zipcode
72 FROM
73     theater;
74
75
76 /*
77     Query to view all theaters that are currently playing the
78     Avengers movie.
```

```
79 */
80 SELECT DISTINCT
81     name
82 FROM
83     theater,
84     showtime,
85     movie
86 WHERE
87     movie.title LIKE '%Avengers%'              AND
88     movie.movie_id=showtime.movie_id           AND
89     showtime.theater_id=theater.theater_id;
90
91
92 /*
93     Query to view the theater name, show start time, show date,
94     and ticket price for all theaters playing the Avengers movie.
95 */
96 SELECT
97     name,
98     start_time,
99     show_date,
100    price
101 FROM
102    theater,
103    movie,
104    showtime,
105    ticket
106 WHERE
107    movie.title LIKE '%Avengers%'                AND
108    showtime.movie_id=movie.movie_id             AND
109    showtime.theater_id=theater.theater_id       AND
110    ticket.showtime_id=showtime.showtime_id
111 ORDER BY
112    theater.name,
113    show_date,
114    start_time;
115
116
117 /*
118    Query to view the name of all theaters, movie titles,
119    show start time, show date, and ticket price for all
120    theaters that have any showtimes.
121 */
122 SELECT
123    name,
124    title,
125    start_time,
126    show_date,
127    price
128 FROM
129    theater,
130    movie,
131    showtime,
132    ticket
133 WHERE
134    showtime.movie_id=movie.movie_id             AND
```

```
135       showtime.theater_id=theater.theater_id       AND
136       ticket.showtime_id=showtime.showtime_id
137   ORDER BY
138       theater.name,
139       showtime.show_date,
140       showtime.start_time;
141
142
143   /*
144       Query to view the title of all movies that were
145       produced and/or directed by Steven Spielberg
146   */
147   SELECT DISTINCT
148       title    AS "Spielberg Film"
149   FROM
150       movie,
151       director,
152       producer
153   WHERE
154       movie.director_id=director.director_id       AND
155       director.name='Steven Spielberg'             OR
156       movie.producer_id=producer.producer_id       AND
157       producer.name='Steven Spielberg';
158
159
160   /*
161       Query to view all theater names, the types of projectors
162       the theater uses, as well as the number of each project
163       the theater uses.
164   */
165   SELECT
166       name                      AS "Theater",
167       projector_type            AS "Projector",
168      COUNT(projector_type)      AS "Count"
169   FROM
170       theater,
171       showroom
172   WHERE
173       showroom.theater_id=theater.theater_id
174   GROUP BY
175       name,
176       projector_type
177   ORDER BY
178       name,
179       projector_type;
180
181
182   /*
183       Query to view the number of tickets sold, ticket sales, and the number
184       of tickets that were redeemed for each theater.
185   */
186   SELECT
187       name                                                    AS "Theater",
188       COUNT(ticket)                                           AS "Tickets
              Sold",
189       SUM(ticket.price)                                       AS "Sales ($)",
```

```
190     COUNT(CASE WHEN ticket.wasUsed='t' THEN 1 ELSE NULL END)     AS "Ticket
            Redeemed"
191 FROM
192     ticket,
193     showtime,
194     theater
195 WHERE
196     ticket.showtime_id=showtime.showtime_id AND
197     showtime.theater_id=theater.theater_id
198 GROUP BY
199     name;
200
201
202 /*
203     Query to view the number of tickets sold, the number of tickets available,
204     for each showtime.
205 */
206 SELECT
207     showtime.showtime_id                    AS "Showtime",
208     showtime.show_date                      AS "Date",
209     showtime.start_time                     AS "Start Time",
210     movie.title                             AS "Movie",
211     COUNT(ticket)                           AS "Tickets Sold",
212     showroom.capacity-COUNT(ticket)         AS "Tickets Available"
213 FROM
214     ticket,
215     showtime,
216     showroom,
217     movie
218 WHERE
219     ticket.showtime_id=showtime.showtime_id     AND
220     showtime.showroom_id=showroom.showroom_id   AND
221     showtime.movie_id=movie.movie_id
222 GROUP BY
223     showtime.showtime_id,
224     showtime.show_date,
225     showtime.start_time,
226     showroom.capacity,
227     movie.title
228 ORDER BY
229     showtime.showtime_id,
230     showtime.show_date,
231     showtime.start_time;
```

11.2 Sample Transaction

```
1 /*
2         Creates a new theater with 2 showrooms.
3 */
4 BEGIN TRANSACTION;
5         INSERT INTO theater VALUES (DEFAULT, 'Marquee Cinemas: Southridge 12',
              'Charleston', 'WV', '25309');
6         INSERT INTO showroom VALUES (22, lastval(), 'IMAX', 55);
7         INSERT INTO showroom VALUES (24, lastval(), 'LASER', 70);
8 END TRANSACTION;
9
```

```
10
11 /*
12         Adds a new ticket to a showtime.
13 */
14 BEGIN TRANSACTION;
15         INSERT INTO ticket VALUES (lastval(), DEFAULT, 12.50, now(), now(),
               FALSE);
16 END TRANSACTION;
17
18
19 /*
20         Add a new movie.
21 */
22 BEGIN TRANSACTION;
23         INSERT INTO      director VALUES (DEFAULT, "Christopher Nolan");
24         INSERT INTO producer VALUES (DEFAULT, "Kevin De La Noy");
25         INSERT INTO movie VALUES (DEFAULT, "The Dark Knight Rises", NULL, 2012,
               "Action", lastval(), lastval());
26 END TRANSACTION;
27
28
29 /*
30         Updates the ticket price for IMAX movies.
31 */
32 BEGIN TRANSACTION;
33         UPDATE
34                 ticket, showtime, showroom
35         SET
36                 ticket.price = 13.75;
37         WHERE
38                 ticket.showtime_id=showtime.showtime_id          AND
39                 showtime.showroom_id=showroom.showroom_id        AND
40                 showroom.projector_type="IMAX";
41 END TRANSACTION;
42
43
44 /*
45         Delete all showtimes for Avengers movies.
46 */
47 BEGIN TRANSACTION;
48         DELETE FROM
49                 ticket
50         USING
51                 showtime, movie
52         WHERE
53                 ticket.showtime_id=showtime.showtime_id AND
54                 showtime.movie_id=movie.movie_id                AND
55                 movie.ticket LIKE '%Avengers%'
56
57         DELETE FROM
58                 showtime
59         USING
60                 movie
61         WHERE
62                 showtime.movie_id=movie.movie_id        AND
63                 movie.ticket LIKE '%Avengers%'
```

```
64 END TRANSACTION;
```

12 Summary of Revisions

Some critiques were made by our fellow Computer Science colleagues, namely Shawn Cheeks and Jonathan McQuery, regarding use case diagrams as well as SQL code. For example, it was suggested that we cut down on the number of joins being used in SQL queries to ensure maximum performance. Furthermore, with regard to the physical data model, recommendations were made to keep in mind how often a certain table may be accessed. Finally, a recommendation was made to elaborate further on the Identification of Data Tasks. Each suggestion was taken and acted upon resulting in the final version presented.

13 Trunitin.com - *REDACTED*

14 Metacognitive Reflection

1. Did I solve the right problem?

   We were asked to design a database application that is data intensive and would be beneficial to users. We designed an application that would be usable for movie theatres wishing to draw in a larger demographic. Our application enables users to find out information about any movie they desire including see where it is showing, what time it is being shown, and buying tickets for that movie. It also allows them to find movies by director, actor/actress, producer, title, and genre which will help them narrow down their search for a movie that suits their desires. Employees can also easily update showtimes, ticket prices, and check for valid prices. Lastly, corporate employees have the option to add new theatre locations as the chain expands. Since our database pulls information from the IMDB database, enables users to query and manipulate that data, and conceptually allows them to expand their business as well as gather information, it is not only data intensive but also beneficial. As such, I would say we solved the right problem.

2. Did I solve the problem right?

   As stated in the previous question, we met all of the requirements in making our database design data intensive and beneficial. We provided conceptual functionality for a database that would benefit not only the users but also the employees of the corporation and local theatres as well. With it relying heavily on IMDB, we have in fact designed a database that is data intensive. Therefore, we solved the problem that was placed before us correctly.

3. How did I approach solutions to the problems?

   We worked together and split the work evenly. This made work much easier and less time consuming. We relied on each other for advice and answers to any concerns we had about the dsesign.

4. What strategies and techniques did I draw upon?

We drew upon our problem solving techniques and communication skills. These techniques were extremely beneficial in designing this application.

5. Did I learn a new strategy in completing this assignment? If so, how is it different from and similar to the repertoire of techniques that I have already acquired?

   We learned how to work together as a team and divide work evenly. It is very similar to past projects in that we have worked in teams before and have had to divide work, but in this case, everyone pulled their own weight, which was highly beneficial.

6. Any other information you may wish to add $\cdots$

   We wish to add no other information.

15 Self-assessment

You need to assign a grade for this assignment yourself. Use the rubric listed below to come up with a score. The instructor will also assign a score. Without this section, assignment will be returned with a score of 0.

The first two traits correspond to writing and the remaining ones relate to domain aspects of the project.

Perf Level Trait	Poor	Fair	Good	Outstanding
Diction	Chooses non-technical vocabulary that inadequately conveys the intended meaning of the communication.	Chooses technical vocabulary that conveys the intended meaning of the communication.	Chooses appropriate, technical, and varied vocabulary that conveys the intended meaning of the communication.	Chooses lively, precise, technical, and compelling vocabulary and skillfully communicates the message.
Communication Style	Has only a few (but noticeable) errors in style, mechanics, or other issues that might distract from the message.	Is virtually free of mechanical, stylistic or other issues.	Uses complex and varied sentence styles, concepts, or visual representations.	Creates a distinctive communication style by combining a variety of materials, ideas, or visual representations.
Application Selection	Not a data-intensive application.	Application is somewhat data-intensive	Application is data-intensive but limited access to domain expertise.	Application is data-intensive with adequate access to domain expertise.
Use-cases	Less than 50% of the use-cases are identified, and documented poorly.	Over 75% of the uses-cases are identified and documented using a standard template.	All the use-cases are identified, but detail is missing for some use-cases.	All the use-cases are identified, well-documented using a standard template, and verified against application requirements.
Data Tasks	Inputs, outputs, and possible error conditions are documented for less than 50% of data tasks.	Inputs, outputs, and possible error conditions are documented for less than 75% of data tasks.	Inputs, outputs, and possible error conditions are documented for all data tasks.	Inputs, outputs, and possible error conditions are documented for all data tasks. Processing logic (or high-level algorithms) for transforming inputs into outputs is also described.
Transactions and Queries	Less than 50% of the transactions and queries are identified and described.	Less than 75% of the transactions and queries are identified and described.	All the transactions and queries are identified and described.	All the transactions and queries are identified and described including their frequency of execution.

Perf Level Trait	Poor	Fair	Good	Outstanding
Data Models	Only conceptual data model is described in detail. Cursory treat of logical data model. Physical data model design is missing.	Conceptual and logical data models are described in detail. Physical data model design is missing.	Conceptual, logical, and physical data models are described completely and precisely.	Conceptual, logical, and physical data models are described completely and precisely. Database normalization based on functional dependencies is discussed in detail.
Creation and Loading	SQL scripts are written and executed to create the database and load the data. Data in the database is trivial in size.	SQL scripts are written and executed to create the database and load the data. Data in the database is moderate in size.	Conceptual, logical, and physical data models are described completely and precisely. Data in the database is huge in size – in the order of millions of rows.	Conceptual, logical, and physical data models are described completely and precisely. Data in the database is huge in size – in the order of millions of rows. Detail evidence is provided on how referential integrity constraints are resolved.
Implementing Transactions and Queries	Less than 50% of the transactions and queries are implemented.	Less than 75% of the transactions and queries are implemented.	All the transactions and queries are implemented; run and execute correctly.	All the transactions and queries are implemented; run and execute correctly. There is also written evidence that transactions and queries are tested.
Revisions	Only peer or instructor feedback is solicited, but not incorporated.	Both peer and instructor feedback is solicited but not incorporated.	Both peer and instructor feedback is solicited and incorporated.	Both peer and instructor feedback solicited and incorporated. Evidence is presented to show how the feedback improved the document.
Turnitin.com	No submission is made to turnitin.com	Made to turnitin.com but results are not analyzed.	Made to turnitin.com and results are cursorily analyzed.	Made to turnitin.com and results are analyzed thoroughly.
Meta-cognitive Reflection	Not performed.	Is shallow and incomplete.	Is complete but not thorough.	Is complete and thorough.

Use the following table to score your solution. Circle the appropriate number in each row. For example, to circle 4, use the LaTeX markup code `\circled{4}`, which produces (4).

Perf Level / *Trait*	*Poor*	*Fair*	*Good*	*Outstanding*
Diction	2	3	4	(5)
Communication Style	2	3	4	(5)
Application Selection	4	6	8	(10)
Use-cases	4	6	8	(10)
Data Tasks	4	6	8	(10)
Transactions and Queries	4	6	8	(10)
Data Models	4	6	8	(10)
Creation and Loading	4	6	8	(10)
Implementing Transactions and Queries	4	6	8	(10)
Revisions	4	6	8	(10)
Turnitin.com	2	3	4	(5)
Meta-cognitive Reflection	2	3	4	(5)

Total score: 100 / 100.