Project Report

Database Associates

Vit Bubak
Lian Duan
Ray Hylock
Todd Papke
Table of Contents

Note the mapping of each chapter to the specified page(s).

<table>
<thead>
<tr>
<th>Chapters</th>
<th>Summary of the content for the chapter</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: Requirements Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Introduction and Basic Requirement</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.1 Identification of Different Types of Users</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1.2 Basic Queries</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2: Revised Version of the Conceptual Schema and Data Dictionary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.1 Revised Version of Conceptual Schema</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2.2 Data Dictionary</td>
<td>5 - 9</td>
</tr>
<tr>
<td></td>
<td>2.3 Data Dictionary – Cardinality Constraints Information</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 3: Revised Version of Relational Schema and Data Dictionary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.1 Revised Version of Relational Schema</td>
<td>11 - 16</td>
</tr>
<tr>
<td></td>
<td>3.2 Alternative Designs for the Subclasses</td>
<td>17 - 19</td>
</tr>
<tr>
<td></td>
<td>3.3 Data Dictionary</td>
<td>20 - 26</td>
</tr>
<tr>
<td></td>
<td>3.A Appendix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- SQL Statements to Create Tables and Define Constraints</td>
<td>27 - 31</td>
</tr>
<tr>
<td></td>
<td>- Triggers and Procedures Related to the Tables</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 4: Data Population and Queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.1 Data Population</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>4.2 Queries</td>
<td>32 - 34</td>
</tr>
<tr>
<td>Chapter 5: Triggers and Procedures</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.1 Triggers</td>
<td>35 - 36</td>
</tr>
<tr>
<td></td>
<td>5.2 Procedures</td>
<td>36 - 37</td>
</tr>
<tr>
<td></td>
<td>Included is this chapter implementation of two triggers and one procedure. Explained, in nat. language is the functionality of the triggers, and why they are helpful. The procedure Uses a cursor. The documentation is understandable by anyone unfamiliar to the project.</td>
<td></td>
</tr>
<tr>
<td>Chapter 6: Interface and Reports</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Illustration of a Web-Interface for Client/User Interaction</td>
<td>38 - 48</td>
</tr>
<tr>
<td></td>
<td>Results of the Queries Written for Chapter 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Included is this chapter is a description of (beta version) of the web interface developed for the database. We note that the queries are discussed in this chapter along with the description of the web interface and hence the chapter is not partitioned in sub-chapters.</td>
<td></td>
</tr>
<tr>
<td>Chapter 7: Conclusions and Implementation Plan</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Implementation Plan and Conclusion</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>7.A Appendix</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Contract Estimate Summary Option 1</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>- Contract Estimate Summary Option 2</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>In this final chapter, we describe the steps needed to implement the project on a real-world database management system (presuming the implementing consultant has our report and design easily available). We also include approximate estimates of person-hours (time) and hardware/software costs. A tabular layout summarizing our estimates is also included.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1  Requirements Analysis

Introduction

While the IMDb (http://www.imdb.com/) movie database serves as a useful repository of movie information, it’s use as a source for aggregate movie reviews is limited. While sites like Rotten Tomatoes (http://www.rottentomatoes.com/) serve as a community portal for reviewers to come together as a community and collectively rate movies, they fall short in their ability to allow the user to quickly track the contributing artists that are part of the movie production (i.e. directors, actors, producers, etc.). Additionally, box office receipts and weekly standings aren’t a component of either, but remain the focus of sites such as Hollywood Reporter (http://www.hollywoodreporter.com/hr/index.jsp).

In order to create a more comprehensive site for EverythingMovies, the database schema must be comprehensive enough to allow for multiple simultaneous queries (generated from HTML user forms through a JSP tag library architecture) through a “round-robin” JDBC connection pool, while still allowing for real time updates and contributions by the user community. Also, the schema must be designed to allow for table abstraction across a hardware topology with an index that exists upon its own network server (again for ease of scalability across a server topology as the connection pool grows to accommodate the anticipated user community). Our intent is to make the database public domain with a Creative Commons usage license. The license will allow for reuse as long as there is a click-through “EverythingMovies” brand icon present on the web site that makes use of our database engine. EverythingMovies will utilize a click-thru revenue sharing scheme as the primary revenue model.

While the Oracle DB architecture has historically proven to be scalable through a variety of software and hardware optimization strategies, we also recognize that utilization of a schema that infers Oracle exclusivity may not be in the best interest(s) of the adapting user community that we want to attract with our data offering. Therefore, every attempt will be made to homogenize the SQL in order to allow for data loading into other database engines, specifically PostgreSQL and MySQL. The initial POC effort may use one of these “open source” database engines as necessary due to budgetary constraints.

Basic Requirements

The goal of our client is to create a new type of movie web site that is more comprehensive than the popular ones that currently are available on the Internet.

The primary goal of our effort is to create a “proof of concept” system that could serve as a prototype for illustrating the concepts to potential Venture Capital funding sources. Additionally, the database schema will be used to iron out potential scalability issues that might arise if views are required that weren’t considered during database design (this could arise, for instance, if the initial VC round comes with functional considerations that were outside the initial scope of the project).
Chapter 1.1 Identification of the Different Types of Users and Queries

In this section, we identify three different types of users to whom our database is directed other than the database administrators. We also state what information (given the relations identified above) should be accessible to each user. In addition, we write five SQL queries to address the needs of these users.

The three different types of users that will use our database are:
1. Clients of the database - people who want to use the database to find information about movies. This group can be potentially divided in (a) casual clients who would search the database for any (common) information about the movies, their actors, directors, awards won, et cetera and (b) specialized clients with more complicated search requests. In either case, the implementation of the query interface is the same for most of the users.
2. Contributors to the database - people who are going to add new information to the database. These people differ from the system administrators in that they only add information based on prespecified constraints.
3. System administrators - people who will manage and upgrade/alter/program the database. This group, however, is – in every sense – the same as the database administrators.

Chapter 1.2 Basic Queries

Below, we include examples of the queries that the two basic types of users of the database (defined as clients and contributors above) might find useful. (Note that the examples of the queries given are further discussed in Chapter 4 (Queries) and the results of the queries are given in Chapter 6 (Interface and Reports).

1. Clients of the database - queries

Query CL1) Given a specific time period, a user might be interested in finding the top (three, five, ten) box office movies aired in a given region (say, North America).

Query CL2) Given actor's/actress' name, a user might want to find the find the movies that the actor/actress acts in.

Query CL3) Given the title of the movie (e.g., “Titanic”), a user might want to find out who are the cast members for the movie specified or, what are the awards won by a given movie.

Query CL4) At other times, a user might want to search all the awards won by a given movie.

2. Contributors of the database - queries

Query CO1) A contributor might want insert, update, or delete a movie or a show to/from the database (this query is illustrated in Chapter 6, page 46 of this report).

Query CO2) A contributor might want insert or update the database with any other relevant information.
Chapter 2. Revised Version of the Conceptual Schema and Data Dictionary
<table>
<thead>
<tr>
<th>Schema Construct</th>
<th>Construct Description</th>
<th>Other Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEOPLE</td>
<td>Entity class, to model persons involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Superclass to the following Subclasses <em>(overlapping)</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACTORS, DIRECTORS, WRITERS, PRODUCERS, COMPOSERS, and EDITORS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>ternary</em> Relationship class Won_by (PEOPLE:AWARD_INSTANCES:SHOWS)</td>
<td>Cardinality [↓]</td>
</tr>
<tr>
<td></td>
<td>• PersonID Identifying number of the person</td>
<td>primary identifier</td>
</tr>
<tr>
<td></td>
<td>• DOB Date of birth of the person</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Age Age of the person</td>
<td>derived attribute</td>
</tr>
<tr>
<td></td>
<td>• FName First name of the person</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• mName Middle name of the person</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• lName Last name (surname) of the person</td>
<td></td>
</tr>
<tr>
<td>– ACTORS</td>
<td>Entity class, to model actors involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Earn {ACTORS:SHOWS} with one derived Weak class</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Act_in {ACTORS:SHOWS} with one derived Weak class</td>
<td>cardinality [0:1]</td>
</tr>
<tr>
<td></td>
<td>• screenFN First (screen) name of the actor</td>
<td>multivalued attr.</td>
</tr>
<tr>
<td></td>
<td>• screenMN Middle (screen) name of the actor</td>
<td>multivalued attr.</td>
</tr>
<tr>
<td></td>
<td>• screenLN Last (screen) name of the actor</td>
<td>multivalued attr.</td>
</tr>
<tr>
<td></td>
<td>Act_in Relationship that models the actor act in shows</td>
<td></td>
</tr>
<tr>
<td>– DIRECTORS</td>
<td>Entity class, to model directors involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Direct {DIRECTORS:SHOWS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>Write Relationship that models the writer write shows</td>
<td></td>
</tr>
<tr>
<td>– WRITERS</td>
<td>Entity class, to model writers (screenwriters) involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Write {WRITERS:SHOWS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>Produce Relationship that models the producer produce shows</td>
<td></td>
</tr>
<tr>
<td>– PRODUCERS</td>
<td>Entity class, to model producers involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Produce {PRODUCERS:SHOWS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>Compose Relationship that models the composer compose shows</td>
<td></td>
</tr>
<tr>
<td>– COMPOSERS</td>
<td>Entity class, to model composers involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Compose {COMPOSERS:SHOWS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>Edit Relationship that models the editor edit shows</td>
<td></td>
</tr>
<tr>
<td>– EDITORS</td>
<td>Entity class, to model editors involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Edit {EDITORS:SHOWS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td>Schema Construct</td>
<td>Construct Description</td>
<td>Other Information</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><strong>Edit</strong></td>
<td>Relationship that models the editor edit shows</td>
<td></td>
</tr>
<tr>
<td>ACT_HISTORY</td>
<td>Weak Entity class, to model costumers who (actually) travel</td>
<td>Existence defined by Relationship Class Act_in {ACTORS:SHOWS}, cardinality [1:1]</td>
</tr>
<tr>
<td>• role</td>
<td>Actors role in the film/show</td>
<td></td>
</tr>
<tr>
<td>SALARIES/POINTS</td>
<td>Weak Entity class, to model the salaries and points-based earnings of the actors</td>
<td>Existence defined by Relationship Class Earn {ACTORS:SHOWS}, cardinality [1:M]</td>
</tr>
<tr>
<td>• salID</td>
<td>Identifying number for the salary</td>
<td>primary identifier</td>
</tr>
<tr>
<td>– SALARIES</td>
<td>Weak Entity Subclass, to model the salaries of the actors</td>
<td>Existence defined by Superclass + Relationship Class Earn {ACTORS:SHOWS}</td>
</tr>
<tr>
<td>• amount</td>
<td>Salary for the actor</td>
<td></td>
</tr>
<tr>
<td>– POINTS</td>
<td>Weak Entity Subclass, to model the points-based earnings of the actors</td>
<td>Existence defined by Superclass + Relationship Class Earn {ACTORS:SHOWS}</td>
</tr>
<tr>
<td>• points</td>
<td>Number of points earned by the show</td>
<td></td>
</tr>
<tr>
<td>• value</td>
<td>Derived value of the points that forms part of actor’s total salary</td>
<td>secondary ident.</td>
</tr>
<tr>
<td><strong>Earn</strong></td>
<td>Relationship that models how much the actor earns in the show</td>
<td></td>
</tr>
<tr>
<td>AWARDS</td>
<td>Entity class + Typing class, to model actor/film/show awards</td>
<td>Derived Subtype: AWARD_INSTANCES, cardinality [1:1]</td>
</tr>
<tr>
<td>• awardID</td>
<td>Identifying number for the award</td>
<td>primary identifier</td>
</tr>
<tr>
<td>• name</td>
<td>Name of the award (e.g. Academy Awards)</td>
<td></td>
</tr>
<tr>
<td>ORGANIZATIONS</td>
<td>Entity class, to model organizations that award the films/shows awards</td>
<td>Existence defined by Relationship class Hand_out {AWARDS:ORGANIZATIONS}, cardinality [1:M]</td>
</tr>
<tr>
<td>• orgID</td>
<td>Identifying number for the organization</td>
<td>primary identifier</td>
</tr>
<tr>
<td>• name</td>
<td>Name of the organization awarding the award (e.g. Film Academy)</td>
<td></td>
</tr>
<tr>
<td><strong>Hand_out</strong></td>
<td>Relationship that models the organization hands out awards</td>
<td></td>
</tr>
<tr>
<td>AWARD_INSTANCES</td>
<td>Entity class, to model the types of awards given to actors/films/shows</td>
<td>Supertype is AWARD_INSTANCES Typing Class, cardinality [↓]</td>
</tr>
<tr>
<td>• aiID</td>
<td>Identifying the particular type of the award with a number</td>
<td>primary identifier</td>
</tr>
<tr>
<td>• date</td>
<td>Date the award was won (awarded)</td>
<td>photo</td>
</tr>
<tr>
<td>Schema Construct</td>
<td>Construct Description</td>
<td>Other Information</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>DISTRIBUTORS</td>
<td>Entity class, to model the distributors for the shows/films</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Distribute {DISTRIBUTORS:SHOWS}</td>
<td>cardinality [0:M]</td>
</tr>
<tr>
<td></td>
<td>• distID Distributor for a film-movie</td>
<td>primary identifier</td>
</tr>
<tr>
<td></td>
<td>• name Name of the distributor</td>
<td></td>
</tr>
<tr>
<td>Distribute</td>
<td>Relationship that models the distributor distribute shows</td>
<td></td>
</tr>
<tr>
<td>RATINGS</td>
<td>Entity class, to model the distributors for the shows/films</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Receive {RATINGS:SHOWS}</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>• rateID Identifying number for the rating</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• rating Rating of the show/film</td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Relationship that models the show’s rating.</td>
<td>[see below]</td>
</tr>
<tr>
<td>RATING HISTORY</td>
<td>Weak Entity class, to model the histories of ratings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Existence defined by Relationship Receive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• rHist Identifying code for the rating history</td>
<td>partial identifier</td>
</tr>
<tr>
<td>SHOWS</td>
<td>Entity class, to model shows/films</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Superclass to the following Subclasses (<em>partition</em>)</td>
<td></td>
</tr>
<tr>
<td>FILMS, SHOWS</td>
<td>In: <em>ternary</em> Relationship class Won_by {SHOWS:AWARD_INSTANCES:PEOPLE}</td>
<td>cardinality [*↓+]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Earn {SHOWS:ACTORS}</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Act_in {SHOWS:ACTORS}</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Direct {SHOWS:DIRECTORS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Write {SHOWS:WRITERS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Produce {SHOWS:PRODUCERS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Compose {SHOWS:COMPOSERS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Edit {SHOWS:EDITORS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Distribute {SHOWS:DISTRIBUTORS}</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Receive {SHOWS:RATINGS}</td>
<td>cardinality [0:M]</td>
</tr>
<tr>
<td></td>
<td>In: <em>binary</em> Relationship class Generate {SHOWS: COUNTRY_GROUPS} with one derived Weak class</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>• showID Identifying number for the show/film</td>
<td>primary identifier</td>
</tr>
<tr>
<td></td>
<td>• title Name of the show/film</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• rating Rating of the show/film</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• language Language of the show/film</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• genre Genre of the show/film</td>
<td></td>
</tr>
<tr>
<td>FILMS</td>
<td>Entity class, to model writers (screenwriters) involved in films/shows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subclass to Superclass PEOPLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Derived from Aggregate Class COLLECTIONS</td>
<td>cardinality [0:1]</td>
</tr>
<tr>
<td>Schema Construct</td>
<td>Construct Description</td>
<td>Other Information</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>filmID</td>
<td>Identifying number for the show/film</td>
<td>primary identifier</td>
</tr>
<tr>
<td>year</td>
<td>Name of the show/film</td>
<td></td>
</tr>
<tr>
<td>runtime</td>
<td>Rating of the show/film</td>
<td></td>
</tr>
<tr>
<td>TV_SHOWS</td>
<td>Subclass to Superclass SHOWS + Typing class, to model the TV shows</td>
<td></td>
</tr>
<tr>
<td>tvshowID</td>
<td>Identifying number for the TV show</td>
<td>primary identifier</td>
</tr>
<tr>
<td>startDate</td>
<td>The date the TV show started airing</td>
<td></td>
</tr>
<tr>
<td>endDate</td>
<td>The date the TV show ended airing</td>
<td></td>
</tr>
<tr>
<td>EPISODES</td>
<td>Entity class, to model shows/films episodes</td>
<td></td>
</tr>
<tr>
<td>episodeID</td>
<td>Identifying number for the episode</td>
<td>primary identifier</td>
</tr>
<tr>
<td>title</td>
<td>Name of the episode</td>
<td></td>
</tr>
<tr>
<td>relDate</td>
<td>Date the episode was aired</td>
<td></td>
</tr>
<tr>
<td>COLLECTIONS</td>
<td>Aggregate entity class to FILMS</td>
<td>cardinality [2:M]</td>
</tr>
<tr>
<td>collID</td>
<td>Identifying number for the collection of film/show</td>
<td>primary identifier</td>
</tr>
<tr>
<td>colName</td>
<td>Name for the collection</td>
<td></td>
</tr>
<tr>
<td>bonFeat</td>
<td>Bonus features coming with the collection</td>
<td></td>
</tr>
<tr>
<td>COUNTRIES</td>
<td>Entity class, to model the countries (of origin of the shows/movies)</td>
<td></td>
</tr>
<tr>
<td>countryID</td>
<td>Identifying number for the country</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name of the country</td>
<td></td>
</tr>
<tr>
<td>Make_up</td>
<td>Models the countries belonging to a given country group</td>
<td></td>
</tr>
<tr>
<td>COUNTRY_GROUPS</td>
<td>Aggregate entity class to COUNTRIES to model the distrib’s for the shows/films</td>
<td>cardinality [1:M]</td>
</tr>
<tr>
<td>cgID</td>
<td>Identifying number for the country group</td>
<td>primary identifier</td>
</tr>
<tr>
<td>startDate</td>
<td>Date when the country becomes a member of a group</td>
<td></td>
</tr>
<tr>
<td>endDate</td>
<td>Date when the country ends being a member of a group</td>
<td></td>
</tr>
<tr>
<td>REVENUE_HISTORY</td>
<td>Weak Entity class, to model the revenue history for the films</td>
<td></td>
</tr>
<tr>
<td>revhID</td>
<td>Identifying number for the revenues</td>
<td>primary identifier</td>
</tr>
<tr>
<td>amount</td>
<td>Amount of revenues</td>
<td></td>
</tr>
<tr>
<td>timePer</td>
<td>Time period for the revenues</td>
<td></td>
</tr>
<tr>
<td>Recorded_in</td>
<td>Relationship that models the revenue in which currency</td>
<td></td>
</tr>
<tr>
<td>CURRENCIES</td>
<td>Entity class, to model the currencies (in which the revenues are denominated)</td>
<td></td>
</tr>
</tbody>
</table>

In: binary Relationship class Recorded_in {REVENUE_HISTORY:CURRENCIES} cardinality [1:M]
<table>
<thead>
<tr>
<th>Schema Construct</th>
<th>Construct Description</th>
<th>Other Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>• curID</td>
<td>Identifying number for the currency</td>
<td>primary identifier</td>
</tr>
<tr>
<td>• name</td>
<td>Name of the currency</td>
<td></td>
</tr>
<tr>
<td>Experience</td>
<td>Relationship that models the currency fluctuation</td>
<td></td>
</tr>
<tr>
<td>FLUCTUATIONS</td>
<td>Weak Entity class, to model the currencies (in which revenues are denominated)</td>
<td>cardinality [1:1]</td>
</tr>
<tr>
<td></td>
<td>Existence defined by Relationship Experience</td>
<td></td>
</tr>
<tr>
<td>• fluID</td>
<td>Identifying number for the fluctuations</td>
<td>secondary ident.</td>
</tr>
<tr>
<td>• change</td>
<td>Change in the fluctuations</td>
<td></td>
</tr>
<tr>
<td>• date</td>
<td>Date of fluctuations</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2.3 Data Dictionary – Cardinality Constraints Information

Note 1a)
Cardinalities of the binary relationships between two strong classes are included in the data dictionary.

Example: in the binary relationship class Hand_out {ORGANIZATIONS:AWARDS}, included in the description of the Class Entity {ORGANIZATIONS}, the cardinality showed is [1:M], that is one that states how many awards an organization can receive.

Note 1b)
Cardinalities of the binary relationships between one strong and one (derived) weak class follow the same description as Note 1a) and are included in the data dictionary.

Note 2)
Cardinalities of the ternary relationships that include one derived weak class are as follow. Note that, in the data dictionary, the cardinalities for the ternary relationships of this type are described as if the relationships were effectively binary. In those cases, therefore, the cardinalities in the dictionary follow the reasoning as in Note 1)

Each actor can act in zero to many shows CARD-R-CO(Act_in, ACTORS, SHOWS) IN [0:M]
Each show can have from one to many actors CARD-R-CO(Act_in, SHOWS, ACTORS) IN [1:M]

Note 4)
Cardinalities of the ternary relationships are not included directly in the data dictionary. Instead, they are marked by Cardinality [↓] and are discussed below.

Cardinalities in the ternary relationship Won_by {SHOWS:AWARD_INSTANCES:PEOPLE}
Each {SHOWS} and {PEOPLE} combination can win from [0:M] awards
CARD-R-CO(Won_by, SHOWS, AWARD_INSTANCES, PEOPLE) IN [0:M]
Each {SHOWS} and {AWARD_INSTANCES} combination can have from [0:M] people
CARD-R-CO(Won_by, SHOWS, AWARD_INSTANCES, PEOPLE) IN [0:M]
Each {PEOPLE} and {AWARD_INSTANCES} combination win from [0:1] shows
CARD-R-CO(Won_by, SHOWS, AWARD_INSTANCES, PEOPLE) IN [0:1]

Note 5)
Cardinalities of the aggregate class – derived class relationship are also included in the data dictionary.

Example: in the relationship between the aggregate class {COUNTRY_GROUPS} and {COUNTRIES}, the cardinality [1:M] showed at {COUNTRY_GROUPS} in the data dictionary shows the number of countries that can make up a country group. Similarly, cardinality [1:1] shows the number of country groups that a country can belong to (i.e., at least one and at most one!)

Note 6)
Cardinalities of the multivalued attributed not included in the data dictionary are as follows.

Screen names can have 0 to many values: CARD-A(ACTORS, screenFN) IN [0:M]
CARD-A(ACTORS, screenMN) IN [0:M]
CARD-A(ACTORS, screenLN) IN [0:M]

Actors can play one to many roles: CARD-A(ACT_HISTORY, role) IN [1:M]
Chapter 3.1 Revised Version of Relational Schema

- In this section, we convert the revised conceptual schema (shown on p. 3 as an ER diagram) into relational schema in 4NF. We indicate the primary keys, foreign keys, and other integrity constraints. The primary keys are underlined in the schema list. All constraints are also listed in the data dictionary (p. 10 to 16).
- Alternate design(s) for the subclasses (in 4NF) is given on pages 7 to 9. The choice of one design over another is discussed in this section also.
- The functional dependencies of all attributes are also defined. Along the way, we also verify that all relations are in fourth normal form. For relations already in 4NF, we explain briefly how we confirmed it. We normalize relations that are not in 4NF and explain our steps accordingly.

PEOPLE(personID, fName, mName, lName, DOB, age)
F = {personID -> fName, mName, lName, DOB, age}: It is in 4NF since there is only one determinant and transitivity does not exist.

USED OPTION A (1)

ACTORS(personID)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.

SCREEN_NAMES(snid, personID, sfName, smName, slName)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
F = {personID -> sfName, smName, slName}: It is in 4NF since there is only one determinant and transitivity does not exist.

DIRECTORS(personID)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.

WRITERS(personID)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.

PRODUCERS(personID)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.

COMPOSERS(personID)
foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.
EDITORS(personID)

foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE

In 4NF since there are no non-key attributes.

SHOWS(showID, title, genre, language, rating)


F = {showID -> title, genre, language, rating}: It is in 4NF because only one determinant. Using titles would not work since a title can be placed in different genres, have different languages, and different ratings depending up on the country and whether or not it is a remake. This also makes it in 3NF (no transitivity) which is necessary for 4NF.

USED OPTION A (2)

FILMS(showID, runtime, relDate)

foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

check (runtime > 0) or (runtime = ‘NA’)

F = {showID -> runtime, relDate}: In 4NF since there is only one determinant and transitivity does not exist.

TV_SHOWS(showID, startDate, endDate)

foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

F = {showID -> startDate, endDate}: In 4NF since there is only one determinant and transitivity does not exist.

COLLECTIONS(colID)

In 4NF since there are no non-key attributes.

HAVE_FILMS(showID)

foreign key (showID) references FILMS(showID) ON DELETE CASCADE

In 4NF since there are no non-key attributes.

EPISODES(episodeID, showID, title, relDate)

foreign key (showID) references TV_SHOWS(showID) ON DELETE CASCADE

F = {episodeID -> showID, title, relDate}: In 4NF since there is only one determinant and transitivity does not exist since showID (the only likely non-key determinant) cannot uniquely identify any other non-key attribute.

ACT_HISTORY(personID, showID, role)

foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL

foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

In 4NF since there are no non-key attributes.
DIRECT(personID, showID)
foreign key (personID) references DIRECTORS(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WRITE(personID, showID)
foreign key (personID) references WRITERS(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

PRODUCER(personID, showID)
foreign key (personID) references PRODUCERS(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

COMPOSE(personID, showID)
foreign key (personID) references COMPOSERS(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

EDIT(personID, showID)
foreign key (personID) references EDITORS(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

SALARIES_POINTS(personID, showID, salID, type)
foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
F = {personID, showID, salID - > type}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.

USED OPTION A (3)

SALARIES(personID, showID, salID, amount)
foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
foreign key (salID) references SALARIES_POINTS(salID) ON DELETE SET NULL
check (amount >=0.00)
\[ F = \{\text{personID, showID, salID} \rightarrow \text{amount}\}: \text{In } 4\text{NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.} \]

\[ \text{POINTS(personID, showID, salID, points, value)} \]
foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
foreign key (salID) references SALARIES_POINTS(salID) ON DELETE SET NULL
check (points >=0)
check (value>=0.00)
\[ F = \{\text{personID, showID, salID} \rightarrow \text{points, amount}\}: \text{In } 4\text{NF since neither of the non-key attribute can be a determinant and transitivity does not exist because points is not a non-key determinant for amount and vice versa (B} \rightarrow \text{C portion of transitivity).} \]

\[ \text{DISTRIBUTORS(distID, name)} \]
\[ F = \{\text{distID} \rightarrow \text{name}\}: \text{In } 4\text{NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.} \]

\[ \text{DISTRIBUTE(distID, showID)} \]
foreign key (distID) references DISTRIBUTORS(distID) ON DELETE SET NULL
foreign key (showID) references SHOWS(showID) ON DELETE SET NULL
\text{In } 4\text{NF since there are no non-key attributes.} \]

\[ \text{RATINGS(rateID, rating)} \]
check (rating >= 0) and (rating <= 5)
\[ F = \{\text{rateID} \rightarrow \text{rating}\}: \text{In } 4\text{NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.} \]

\[ \text{RECEIVE(showID, rateID)} \]
foreign key (showID) references SHOWS(showID) ON DELETE CASCADE
foreign key (rateID) references RATINGS(rateID) ON DELETE SET CASCADE
\text{In } 4\text{NF since there are no non-key attributes.} \]

\[ \text{COUNTRIES(countryID, name)} \]
\[ F = \{\text{countryID} \rightarrow \text{name}\}: \text{In } 4\text{NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.} \]

\[ \text{COUNTRY\_GROUPS(cgID)} \]
\text{In } 4\text{NF since there are no non-key attributes.}
COUNTRIESMAKE_UP(countryID, cgID)
  foreign key (countryID) references COUNTRIES(countryID) ON DELETE CASCADE
  foreign key (cgID) references COUNTRY_GROUPS(cgID) ON DELETE CASCADE
  F = {countryID \rightarrow cgid}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.

REVENUE_HISTORY(showID, cgID, revID, amount, rhDate)
  foreign key (showID) references SHOWS(showID) ON DELETE CASCADE
  foreign key (cgID) references COUNTRY_GROUPS(cgID) ON DELETE CASCADE
  check (amount \geq 0.00)
  F = {showID, cgID, revID \rightarrow amount, rhDate}: In 4NF since neither of the non-key attribute can be a determinant and transitivity does not exist because amount is not a non-key determinant for rhDate and vice versa (B \rightarrow C portion of transitivity).

CURRENCIES(curlID, name)
  F = {curlID \rightarrow name}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.

RECORDED_IN(showID, cgID, revID, curID)
  foreign key (showID, cgID, revID) references REVENUE_HISTORY(showID, cgID, revID) ON DELETE CASCADE
  foreign key (curID) references CURRENCIES(curlID) ON DELETE CASCADE
  In 4NF since there are no non-key attributes.

FLUCTUATIONS(curlID, fluid, change, flucDate)
  F = {curlID, fluid \rightarrow change, flucDate}: In 4NF since neither of the non-key attribute can be a determinant and transitivity does not exist because change is not a non-key determinant for flucDate and vice versa (B \rightarrow C portion of transitivity).

ORGANIZATIONS(orgID, name)
  F = {orgID \rightarrow name}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.

AWARDS(awardID, name)
  F = {awardID \rightarrow name}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.
AWARD_INSTANCES(aiID, awardID, awardDate)
foreign key (awardID) references AWARDS(awardID) ON DELETE SET NULL
F = {aiID -> awardID, awardDate}: It is in 4NF since there is only one determinant and transitivity does not exist (awardID is only for the type of award).

HAND_OUT(awardID, orgID)
foreign key (awardID) references AWARDS(awardID) ON DELETE SET NULL
foreign key (orgID) references ORGANIZATIONS(orgID) ON DELETE SET NULL
F = {awardID -> orgID}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.

WON_BY_PEOPLE(aiID, personID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_SHOWS(aiID, showID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (showID) references SHOWS(showID) ON DELETE SET NULL
F = {aiID-> showID}: In 4NF since the only non-key attribute cannot be a determinant and transitivity does not exist because there is only one non-key attribute.
Chapter 3.2 Alternative Designs for the Subclasses

For subclass (1): We used option A because there is relationship that involves all of subclass (PEOPLE to AWARD_INSTANCES) and relationships that need each individual subclass and only that particular subclass. Using option B would leave us with many relationships to AWARD_INSTANCES and we would have to add the attributes to each subclass. If we were to use option C, that would require logic to make sure that we had the correct subclass of people (i.e. only ACTORS for the act_in relationship) as well as adding six more attributes to identify the type (we cannot simply use one since we have a cover type [1:M]). In this case, the best alternative would depend on whether you would want to write more code or manage more tables. Our choice would be to use option B since adding more tables would be easier for us to handle than more logic.

OPTION B

<table>
<thead>
<tr>
<th>Table</th>
<th>Attributes</th>
<th>Functional Dependency</th>
<th>Normalization to 4NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTORS</td>
<td>actorID, fName, mName, lName, DOB, age</td>
<td>{actorID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>SCREEN_NAMES</td>
<td>actorID, sfName, smName, slName</td>
<td>{actorID -&gt; sfName, smName, slName}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>DIRECTORS</td>
<td>directorID, fName, mName, lName, DOB, age</td>
<td>{directorID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>WRITERS</td>
<td>writerID, fName, mName, lName, DOB, age</td>
<td>{writerID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>PRODUCERS</td>
<td>producerID, fName, mName, lName, DOB, age</td>
<td>{producerID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>COMPOSERS</td>
<td>composerID, fName, mName, lName, DOB, age</td>
<td>{composerID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
<tr>
<td>EDITORS</td>
<td>editorID, fName, mName, lName, DOB, age</td>
<td>{editorID -&gt; fName, mName, lName, DOB, age}</td>
<td>in 4NF since there is only one determinant and transitivity does not exist.</td>
</tr>
</tbody>
</table>
NEW RELATIONSHIPS TO AWARD_INSTANCES

WON_BY_ACTORS(aiID, actorID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (actorID) references ACTORS(actorID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_DIRECTORS(aiID, directorID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (directorID) references DIRECTORS(directorID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_WRITERS(aiID, writerID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (writerID) references WRITERS(writerID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_PRODUCERS(aiID, producerID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (producerID) references PRODUCERS(producerID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_COMPOSERS(aiID, composerID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (composerID) references COMPOSERS(composerID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

WON_BY_EDITORS(aiID, editorID)
foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL
foreign key (editorID) references EDITORS(editorID) ON DELETE SET NULL
In 4NF since there are no non-key attributes.

For subclass (2): We used option A because, again, we have relationships extending from both the superclass and the subclasses. Using option B would increase the number of relationships by tables by 11 and we would have to add the attributes from SHOWS to the two subclasses. If we did option C, would have to write some logic to help with the aggregate entity class COLLECTIONS (to make sure they were films) and the typing class (for episodes). In this case, since we have a partition, we would only need to add one attribute to SHOWS in order to differentiate between which type (film or episode) the tuple belongs to, but we would have to add all of the attributes from both subclasses, so we will end up with at the minimum, 2 null attributes per record and at most 3. So, for our alternative, we decided to go with option C because a little bit of logic and some empty fields are much easier to program and maintain than 11 additional tables.
OPTION C

SHOWS(showID, title, genre, language, rating, runtime, relDate, startDate, endDate)
F = {showID -> title, genre, language, rating, runtime, relDate, startDate, endDate}: It is in 4NF since there is only one determinant. Using titles would not work as a determinant since a title can be placed in different genres, have different languages, and different ratings depending up on the country and whether or not it is a remake. This also makes it in 3NF (no transitivity) which is necessary for 4NF.

HAVE_FILMS(showID, colID)
foreign key (showID) references SHOWS(showID) ON DELETE CASCADE
foreign key (colID) references COLLECTIONS(colID) ON DELETE CASCADE
In 4NF since there are no non-key attributes.

EPISODES(episodeID, showID, title, relDate)
foreign key (showID) references SHOWS(showID) ON DELETE CASCADE
F = {episodeID -> showID, title, relDate}: It is in 4NF since there is only one determinant and transitivity does not exist (episode titles can be the same, showID’s can be the same, the same for relDate’s).

For subclass (3): Again, we used option A, but this time, it was simply because they were separate themes and had different attributes. Since the superclass is the only one of the three that has a relationship, we would not want to add complexity by using option B. So option C is the simplest alternative.

OPTION C

SALARIES_POINTS(personID, showID, salID, type, amount, points, value)
foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
F = {personID, showID, salID -> type, amount, points, value}: In 4NF since none of the non-key attribute can be a determinant type has two possible values and amount, points, and value can be null) and transitivity does not exist because types is not a non-key determinant for amount/points/value, amount is not a non-key determinant of type/points/value, points is not a non-key determinant of type/amount/value, and value is not a non-key determinant of type/amount/points (B -> C portion of transitivity).
<table>
<thead>
<tr>
<th>Schema Construct</th>
<th>Construct Description</th>
<th>Data Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEOPLE</td>
<td>Relation representing the entity class PEOPLE; stores information on persons involved in films and/or shows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>Primary Key</td>
</tr>
<tr>
<td>• DOB</td>
<td>Date of birth of the person</td>
<td>date</td>
<td>Not Null</td>
</tr>
<tr>
<td>• Age</td>
<td>Age of the person</td>
<td>numeric(3)</td>
<td></td>
</tr>
<tr>
<td>• FName</td>
<td>First name of the person</td>
<td>varchar2(30)</td>
<td>Not Null</td>
</tr>
<tr>
<td>• Mname</td>
<td>Middle name of the person</td>
<td>varchar2(30)</td>
<td></td>
</tr>
<tr>
<td>• Lname</td>
<td>Last name (surname) of the person</td>
<td>varchar2(30)</td>
<td>Not Null</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow FName, mName, Lname, DOB, age</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCREEN_NAME</td>
<td>Relation representing the names of the people; stores information on names of the actors, directors, writers, etc., involved in films/shows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td>• SfName</td>
<td>First (screen) name of the actor</td>
<td>varchar2(30)</td>
<td>Not Null</td>
</tr>
<tr>
<td>• SmName</td>
<td>Middle (screen) name of the actor</td>
<td>varchar2(30)</td>
<td></td>
</tr>
<tr>
<td>• SlName</td>
<td>Last (screen) name of the actor</td>
<td>varchar2(30)</td>
<td>Not Null</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow personID, sfName, smName, slName</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACTORS</td>
<td>Relation representing the entity subclass ACTORS; stores information on actors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow personID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIRECTORS</td>
<td>Relation representing the entity subclass DIRECTORS; stores information on directors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow personID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITERS</td>
<td>Relation representing the entity subclass WRITERS; stores information on writers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow personID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRODUCERS</td>
<td>Relation representing the entity subclass PRODUCERS; stores information on producers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td>FD:</td>
<td>personID \rightarrow personID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPOSERS</td>
<td>Relation representing the entity subclass COMPOSERS; stores information on composers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td></td>
</tr>
<tr>
<td>Relation</td>
<td>Description</td>
<td>Attributes</td>
<td>Checks</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>EDITORS</td>
<td>Relation representing the entity subclass EDITORS; stores information on editors</td>
<td>PersonID: Identifying number of the person</td>
<td>FK (PEOPLE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td>SHOWS</td>
<td>Relation representing the entity class SHOWS; stores information on Films/Shows</td>
<td>showID: Identifying number for the show/film</td>
<td>Primary Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>title: Name of the show/film</td>
<td>varchar2(30)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rating: Rating of the show/film</td>
<td>char(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>language: Language of the show/film</td>
<td>varchar2(15)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>genre: Genre of the show/film</td>
<td>varchar2(15)</td>
</tr>
<tr>
<td>FILMS</td>
<td>Relation representing the sub-class FILMS; stores information on Films</td>
<td>showID: Identifying number for the film</td>
<td>FK (SHOWS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>year: Year when the film was made</td>
<td>Numeric(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime: Year when the film was aired</td>
<td>Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: runtime &gt; 0 or runtime = 'NA'</td>
<td></td>
</tr>
<tr>
<td>TV_SHOWS</td>
<td>Relation representing the sub-class TV_SHOWS; stores information on TV shows</td>
<td>showID: Identifying number for the TV show</td>
<td>FK (SHOWS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>startDate: The date the TV show started airing</td>
<td>Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endDate: The date the TV show ended airing</td>
<td>Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: runtime &gt; 0 or runtime = 'NA'</td>
<td></td>
</tr>
<tr>
<td>COLLECTIONS</td>
<td>Relation representing the Aggregate class COLLECTIONS</td>
<td>colID: Identifying number for the collection of films</td>
<td>Primary Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>colName: Name for the collection</td>
<td>char(10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bonFeat: Bonus features coming with the collection</td>
<td>varchar(20)</td>
</tr>
<tr>
<td>HAVE_FILMS</td>
<td>Relation representing the relationship HAVE_FILMS</td>
<td>colID: Identifying number for the collection of films</td>
<td>FK (COLECS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>showID: Identifying number for the show (film)</td>
<td>FK (SHOWS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: foreign key (showID) references FILMS(showID) ON DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: foreign key (colID) references COLLECTIONS(colID) ON DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td>EPISODES</td>
<td>Relation representing the instantiation class EPISODES; stores information on Episodes</td>
<td>showID: Identifying number for the show (film)</td>
<td>FK (SHOWS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: foreign key (showID) references FILMS(showID) ON DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check constraint: foreign key (colID) references COLLECTIONS(colID) ON DELETE CASCADE</td>
<td></td>
</tr>
<tr>
<td>Schema Construct</td>
<td>Construct Description</td>
<td>Data Type</td>
<td>Constraint</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>• episodeID</td>
<td>Identifying number for the episode</td>
<td>varchar2(10)</td>
<td></td>
</tr>
<tr>
<td>• showID</td>
<td>Identifying number for the show (film)</td>
<td>varchar2(10)</td>
<td></td>
</tr>
<tr>
<td>• title</td>
<td>Name of the episode</td>
<td>varchar2(30)</td>
<td></td>
</tr>
<tr>
<td>• relDate</td>
<td>Date the episode was aired</td>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>

Check constraint: foreign key (showID) references TV_SHOWS(showID) ON DELETE CASCADE
FD: episodeID -> showID, title, relDate

ACT_HISTORY
Relation representing the (weak) entity class ACT_HISTORY; stores information on actors’ acting histories

• personID Identifying number of the person | char(10) | FK (PEOPLE) |
• showID Identifying number of the show/film | char(10) | FK (SHOWS) |

Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

ROLE
Relation representing the entity class ROLE

• personID Identifying number of the person | char(10) | FK (PEOPLE) |
• showID Identifying number of the show/film | char(10) | FK (SHOWS) |
• role Role of an actor in the show/film | varchar2(30) | |

ACT_IN
Relation representing the relationship ACT_IN; stores information on actors playing in shows

• PersonID Identifying number of the person | char(10) | FK (PEOPLE) |
• ShowID Identifying number of the show/film | char(10) | FK (SHOWS) |

Primary key constraint: personID, showID
Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE SET NULL
FD : personID, showID -> personID, showID

DIRECT
Relation representing the relationship DIRECT; stores information on directors playing in shows

• PersonID Identifying number of the person | char(10) | FK (PEOPLE) |
• ShowID Identifying number of the show/film | char(10) | FK (SHOWS) |

Primary key constraint: personID, showID
Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE SET NULL
FD : personID, showID -> personID, showID

WRITE
Relation representing the relationship WRITE; stores information on directors playing in shows

• PersonID Identifying number of the person | char(10) | FK (PEOPLE) |
• ShowID Identifying number of the show/film | char(10) | FK (SHOWS) |
<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
<th>Primary Key Constraint(s)</th>
<th>Foreign Key Constraints</th>
<th>FDs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PRODUCE</strong></td>
<td>Relation representing the relationship PRODUCE</td>
<td>PersonID, showID</td>
<td>personID, showID</td>
<td>personID, showID → personID, showID</td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
<td></td>
</tr>
<tr>
<td>• ShowID</td>
<td>Identifying number of the show/film</td>
<td>char(10)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td><strong>COMPOSE</strong></td>
<td>Relation representing the relationship COMPOSE</td>
<td>PersonID, showID</td>
<td>personID, showID</td>
<td>personID, showID → personID, showID</td>
</tr>
<tr>
<td>• PersonID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
<td></td>
</tr>
<tr>
<td>• ShowID</td>
<td>Identifying number of the show/film</td>
<td>char(10)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td><strong>EDIT</strong></td>
<td>Relation representing the relationship EDIT</td>
<td>personID, showID</td>
<td>personID, showID</td>
<td>personID, showID → personID, showID</td>
</tr>
<tr>
<td>• personID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
<td></td>
</tr>
<tr>
<td>• showID</td>
<td>Identifying number of the show/film</td>
<td>char(10)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td><strong>SALARIES_POINTS</strong></td>
<td>Relation representing the Entity class SALARIES_POINTS; stores information on salaries paid to actors in shows</td>
<td>salID, personID, showID</td>
<td>Primary Key (salID)</td>
<td>salID, personID, showID → type</td>
</tr>
<tr>
<td>• salID</td>
<td>Identifying number for the salary</td>
<td>char(10)</td>
<td>FK (SAL_PTS)</td>
<td></td>
</tr>
<tr>
<td>• personID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
<td></td>
</tr>
<tr>
<td>• showID</td>
<td>Identifying number of the show/film</td>
<td>char(10)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td>• type</td>
<td>Type of salary given out/paid</td>
<td>char(15)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SALARIES</strong></td>
<td>Relation representing the sub-class SALARIES; stores information on salaries</td>
<td>salID, personID, showID</td>
<td>FK (SAL_PTS)</td>
<td></td>
</tr>
<tr>
<td>• salID</td>
<td>Identifying number for the salary</td>
<td>char(10)</td>
<td>FK (SAL_PTS)</td>
<td></td>
</tr>
<tr>
<td>• personID</td>
<td>Identifying number of the person</td>
<td>char(10)</td>
<td>FK (PEOPLE)</td>
<td></td>
</tr>
<tr>
<td>Schema Construct</td>
<td>Construct Description</td>
<td>Data Type</td>
<td>Constraint</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>-----------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>showID</td>
<td>Identifying number of the show/film</td>
<td>char(10)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td>amount</td>
<td>Amount of salary paid</td>
<td>numeric(8,2)</td>
<td>Default 0.00</td>
<td></td>
</tr>
</tbody>
</table>

Check constraint: foreign key (salID) references SALARIES_POINTS(salID) ON DELETE SET NULL + constraints (A) and (B)
Check constraint: amount >=0.00
FD : personID, showID, salID ➔ amount

POINTS | Relation representing the sub-class POINTS; stores information on points derived from shows/films | | |
| salID | Identifying number for the salary | char(10) | FK (SAL_PTS) |
| personID | Identifying number of the person | varchar2(30) | FK (PEOPLE) |
| showID | Identifying number of the show/film | char(5) | FK (SHOWS) |
| points | Number of points earned | numeric(5) | |
| value | Genre of the show/film | numeric(5,2) | Default 0.00 |

Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL
Check constraint: foreign key (showsID) references SHOWS(showsID) ON DELETE SET NULL
Check constraint: points >= 0, value >= 0.00

DISTRIBUTORS | Relation representing the class DISTRIBUTORS; stores information on distributors | | |
| distID | Identification number of the (film/show) distributor | numeric(10) | Primary Key |
| name | Name of the distributor | varchar2(40) | |

FD : distID ➔ name

DISTRIBUTE | Relation representing the relationship DISTRIBUTE | | |
| distID | Identification number of the (film/show) distributor | numeric(10) | FK (DISTRIBS) |
| showID | Identifying number of the show/film | char(5) | FK (SHOWS) |

Check constraint: foreign key (distID) references DISTRIBUTORS(distID) ON DELETE SET NULL
Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

RATINGS | Relation representing the class RATINGS; stores information on ratings | | |
| rateID | Identification number of rating | char(6) | Primary Key |
| ratings | Ratings received by the show/film | varchar2(40) | |

Check constraint: (rating >= 0) and (rating <= 5)
FD : rateID ➔ rating

RECEIVE | Relation representing the relationship RECEIVE | | |
| showID | Identifying number of the show/film | char(10) | FK (SHOWS) |
| rateID | Identifying number of the rating | char(6) | FK (RATINGS) |

Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE CASCADE
Check constraint: foreign key (rateID) references RATINGS(rateID) ON DELETE SET CASCADE

COUNTRIES | Relation representing the class COUNTRIES; | | |
## Database Associates – Internet Movie Database – Project Report – May 9, 2008

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Description</th>
<th>Field</th>
<th>Field Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>stores information on countries from where the shows/films were made</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• countryID</td>
<td>Identifying number of the country where the show/film was made</td>
<td>char(5)</td>
<td>Primary Key</td>
<td></td>
</tr>
<tr>
<td>• name</td>
<td>Name of the country</td>
<td>varchar2(25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD: countryID → name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNTRY GROUPS</td>
<td>Relation representing the aggregate class COUNTRY_GROUPS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• cgID</td>
<td>Identifying number of the country group</td>
<td>char(5)</td>
<td>Primary Key</td>
<td></td>
</tr>
<tr>
<td>• startDate</td>
<td>Date when the country becomes a member of a group</td>
<td>date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• endDate</td>
<td>Date when the country ends being a member of a group</td>
<td>date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNTRIES_MAKE_UP</td>
<td>Relation representing the relationship COUNTRIES_MAKE_UP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• countryID</td>
<td>Identifying number of the country where the show/film was made</td>
<td>char(5)</td>
<td>FK (COUNTR)</td>
<td></td>
</tr>
<tr>
<td>• cgID</td>
<td>Identifying number of the country group</td>
<td>char(5)</td>
<td>FK (CNTRGS)</td>
<td></td>
</tr>
<tr>
<td>Check constraint: (countryID) references COUNTRIES(countryID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check constraint: (cgID) references COUNTRY_GROUPS(cgID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REVENUE_HISTORY</td>
<td>Relation representing the class REVENUE_HISTORY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• showID</td>
<td>Identifying number of the show/film</td>
<td>char(5)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td>• cgID</td>
<td>Identifying number of the country group</td>
<td>char(5)</td>
<td>FK (CNTRGS)</td>
<td></td>
</tr>
<tr>
<td>• revID</td>
<td>Identifying number for the revenues</td>
<td>char(5)</td>
<td>Primary Key</td>
<td></td>
</tr>
<tr>
<td>• amount</td>
<td>Amount of revenues</td>
<td>numeric(12,2)</td>
<td>Default 0.00</td>
<td></td>
</tr>
<tr>
<td>• rhDate</td>
<td>Date for the revenue history</td>
<td>date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check constraint: foreign key (cgID) references COUNTRY_GROUPS(cgID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check constraint: amount &gt;= 0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD: showID, cgID, revID → amount, rhDate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CURRENCIES</td>
<td>Relation representing the class CURRENCIES; stores information on currencies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• curID</td>
<td>Identifying symbol of the currency</td>
<td>char(2)</td>
<td>Primary Key</td>
<td></td>
</tr>
<tr>
<td>• name</td>
<td>Name of the currency</td>
<td>varchar2(20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FD: curID → name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECORDED_IN</td>
<td>Relation representing the relationship RECORDED_IN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• showID</td>
<td>Identifying number of the show/film</td>
<td>char(5)</td>
<td>FK (SHOWS)</td>
<td></td>
</tr>
<tr>
<td>• cgID</td>
<td>Identifying number of the country group</td>
<td>char(8)</td>
<td>FK (CNTRGS)</td>
<td></td>
</tr>
<tr>
<td>• revID</td>
<td>Identifying number for the revenues</td>
<td>char(5)</td>
<td>FK (REV_HIS)</td>
<td></td>
</tr>
<tr>
<td>• curID</td>
<td>Identifying symbol of the currency</td>
<td>char(2)</td>
<td>FK (CURREN)</td>
<td></td>
</tr>
<tr>
<td>Check constraint: foreign key (showID, cgID, revID) references REVENUE_HISTORY(showID, cgID, revID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check constraint: foreign key (curID) references CURRENCIES(curID) ON DELETE CASCADE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLUCTUATIONS</td>
<td>Relation representing the class FLUCTUATIONS; stores information on fluctuations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• curID</td>
<td>Identifying symbol of the currency</td>
<td>char(2)</td>
<td>FK (CURREN)</td>
<td></td>
</tr>
</tbody>
</table>
### Schema Construct

<table>
<thead>
<tr>
<th>Construct Description</th>
<th>Data Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>• fluID Identifying symbol of the currency fluctuation</td>
<td>char(5)</td>
<td>Primary Key</td>
</tr>
<tr>
<td>• change Change in the currency date</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>• flucDate Date of the fluctuation in currency</td>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>

**FD:** curID, fluid → change, fulcDate

#### ORGANIZATIONS

Relation representing the class ORGANIZATIONS; stores information on organizations that give out the awards

- • orgID Identifying symbol of the organization | char(8) | Primary Key |
- • name Name of the organizations | varchar2(35) | |

**FD:** orgID → name

#### AWARDS

Relation representing the (typing) class AWARDS; stores information on awards

- • awardID Identifying symbol of the award | char(8) | Primary Key |
- • name Name of the award | varchar2(30) | |

**FD:** awardID → name

#### AWARD_INSTANCES

Relation representing the instantiation class AWARD_INSTANCES

- • aiID Identifying symbol of the award (instance) | char(8) | |
- • awardID Identifying symbol of the award | char(8) | FK (AWARD) |
- • awardDate Date the award was given out | date | |

Check constraint: foreign key (awardID) references AWARDS(awardID) ON DELETE SET NULL

**FD:** aiID → awardID, awardDate

#### HAND_OUT

Relation representing the relationship HAND_OUT

- • orgID Identifying symbol of the organization | char(8) | FK (ORGAN) |
- • awardID Identifying symbol of the award | char(8) | FK (AWARD) |

Check constraint: foreign key (orgID) references ORGANIZATIONS(orgID) ON DELETE SET NULL

Check constraint: foreign key (awardID) references AWARDS(awardID) ON DELETE SET NULL

#### WON_BY_PEOPLE

Relation representing the relationship WON_BY_PEOPLE

- • aiID Identifying symbol of the award (instance) | char(8) | FK (AW _IN) |
- • personID Identifying number of the person | char(10) | FK (PEOPLE) |

Check constraint: foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL

Check constraint: foreign key (personID) references PEOPLE(personID) ON DELETE SET NULL

#### WON_BY_SHOWS

Relation representing the relationship WON_BY_SHOWS

- • aiID Identifying symbol of the award (instance) | char(8) | FK (AW _IN) |
- • showID Identifying number for the show (film) | char(10) | FK (SHOWS) |

Check constraint: foreign key (aiID) references AWARD_INSTANCES(aiID) ON DELETE SET NULL

Check constraint: foreign key (showID) references SHOWS(showID) ON DELETE SET NULL

Note: In the constraint, we often use shortcuts such as FK in case of Foreign Key or, AW _IN in case of AWARD_INSTANCES.

Also note, that (for the current purposes – and simplicity – we set all the primary keys to a char(10) data type!)
3.A Appendix

SQL Statements to Create Tables and Define Constraints

Below we include the SQL statements that we used to create the tables for our database.

```sql
create table "K272G1"."ACT_HISTORY" (  
  "personID" varchar2(10) CONSTRAINT ACT_HISTORY_personID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,  
  "showID" varchar2(10) CONSTRAINT ACT_HISTORY_showID REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."ACT_IN" (  
  "PersonID" varchar2(10) CONSTRAINT ACT_IN_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,  
  "ShowID" varchar2(10) CONSTRAINT ACT_IN_ShowID REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."ACTORS" (  
  "PersonID" varchar2(10) CONSTRAINT ACTORID REFERENCES PEOPLE("PersonID") ON DELETE CASCADE  
)

create table "K272G1"."AWARD_INSTANCES" (  
  "aiID" varchar2(10) NOT NULL PRIMARY KEY,  
  "awardID" varchar2(10) CONSTRAINT AWARD_INSTANCES_AWARDS_awardID REFERENCES AWARDS("awardID") ON DELETE SET NULL,  
  "awardDate" DATE  
)

create table "K272G1"."AWARDS" (  
  "awardID" varchar2(10) NOT NULL PRIMARY KEY,  
  "name" VARCHAR2(35)  
)

create table "K272G1"."COLLECTIONS" (  
  "colID" varchar2(10) NOT NULL PRIMARY KEY  
)

create table "K272G1"."COMPOSE" (  
  "PersonID" varchar2(10) CONSTRAINT COMPOSE_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,  
  "ShowID" varchar2(10) CONSTRAINT COMPOSE_ShowID REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."COMPOSERS" (  
  "PersonID" varchar2(10) CONSTRAINT COMPOSERID REFERENCES PEOPLE("PersonID") ON DELETE CASCADE  
)

create table "K272G1"."COUNTRIES_MAKE_UP" (  
  "countryID" varchar2(10) CONSTRAINT COUNTRMU_COUNTRIES_countryID REFERENCES COUNTRIES("countryID") ON DELETE CASCADE,  
  "cgID" varchar2(10) CONSTRAINT COUNTRMU_COUNTRY_GROUPS_cgID REFERENCES COUNTRY_GROUPS("cgID") ON DELETE CASCADE  
)
```
create table "K272G1"."COUNTRIES" (  
  "countryID" varchar2(10) NOT NULL PRIMARY KEY,  
  "name" VARCHAR2(25) NOT NULL  
)

create table "K272G1"."COUNTRY_GROUPS" (  
  "cgID" varchar2(10) NOT NULL PRIMARY KEY  
)

create table "K272G1"."CURRENCIES" (  
  "curID" varchar2(10) NOT NULL PRIMARY KEY,  
  "name" VARCHAR2(20)  
)

create table "K272G1"."DIRECT" (  
  "PersonID" varchar2(10) CONSTRAINT DIRECT_PersonID  
    REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,  
  "ShowID" varchar2(10) CONSTRAINT DIRECT_ShowID  
    REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."DIRECTORS" (  
  "PersonID" varchar2(10) CONSTRAINT DIRECTORID  
    REFERENCES PEOPLE("PersonID") ON DELETE CASCADE  
)

create table "K272G1"."DISTRIBUTE" (  
  "distID" varchar2(10) CONSTRAINT DISTRIBUTE_DISTRIBUTORS_distID  
    REFERENCES DISTRIBUTORS("distID") ON DELETE SET NULL,  
  "showID" varchar2(10) CONSTRAINT DISTRIBUTE_SHOWS_showID  
    REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."DISTRIBUTORS" (  
  "distID" varchar2(10) NOT NULL PRIMARY KEY,  
  "name" VARCHAR2(40) NOT NULL  
)

create table "K272G1"."EDIT" (  
  "PersonID" varchar2(10) CONSTRAINT EDIT_PersonID  
    REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,  
  "ShowID" varchar2(10) CONSTRAINT EDIT_ShowID  
    REFERENCES SHOWS("showID") ON DELETE SET NULL  
)

create table "K272G1"."EDITORS" (  
  "PersonID" varchar2(10) CONSTRAINT EDITORID  
    REFERENCES PEOPLE("PersonID") ON DELETE CASCADE  
)

create table "K272G1"."EPISODES" (  
  "episodeID" varchar2(10) NOT NULL PRIMARY KEY,  
  "showID" varchar2(10) NOT NULL CONSTRAINT EPISODES_showID  
    REFERENCES SHOWS("showID") ON DELETE CASCADE,  
  "title" VARCHAR2(30),  
  "reDate" DATE  
)

create table "K272G1"."FILMS" (  
  "showID" varchar2(10) NOT NULL CONSTRAINT FilmsShowID  
    REFERENCES SHOWS("showID") ON DELETE SET NULL,  
  "year" NUMBER(4),  
  "relDate" DATE  
)
```sql
CREATE TABLE "K272G1"."FLUCTUATIONS" (
  "curID" VARCHAR2(10) CONSTRAINT FLUC_CURRENCIES_curID REFERENCES CURRENCIES("curID") ON DELETE CASCADE,
  "fluID" VARCHAR2(10) NOT NULL PRIMARY KEY,
  "change" DATE,
  "flucDate" DATE
)

CREATE TABLE "K272G1"."HAND_OUT" (
  "orgID" VARCHAR2(10) CONSTRAINT HAND_OUT_ORG_orgID REFERENCES ORGANIZATIONS("orgID"),
  "awardID" VARCHAR2(10) CONSTRAINT HAND_OUT_AWARDS_awardID REFERENCES AWARDS("awardID") ON DELETE SET NULL
)

CREATE TABLE "K272G1"."HAVE_FILMS" (
  "colID" VARCHAR2(10) NOT NULL CONSTRAINT HAVE_FILMS_COLECS REFERENCES COLLECTIONS("colID") ON DELETE CASCADE,
  "showID" VARCHAR2(10) NOT NULL CONSTRAINT HAVE_FILMS_showID REFERENCES SHOWS("showID") ON DELETE CASCADE
)

CREATE TABLE "K272G1"."ORGANIZATIONS" (
  "orgID" VARCHAR2(10) NOT NULL PRIMARY KEY,
  "name" VARCHAR2(35)
)

CREATE TABLE "K272G1"."PEOPLE" (
  "PersonID" VARCHAR2(10) NOT NULL PRIMARY KEY,
  "DOB" DATE NOT NULL,
  "Age" NUMBER(3),
  "Fname" VARCHAR2(30) NOT NULL,
  "Mname" VARCHAR2(30),
  "Lname" VARCHAR2(30) NOT NULL
)

CREATE TABLE "K272G1"."POINTS" (
  "salID" VARCHAR2(10) CONSTRAINT POINTS_SAL_PTS REFERENCES SALARIES_POINTS("salID") ON DELETE SET NULL,
  "personID" VARCHAR2(10) CONSTRAINT POINTS_PEOPLE_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,
  "showID" VARCHAR2(10) CONSTRAINT POINTS_showID REFERENCES SHOWS("showID") ON DELETE SET NULL,
  "points" NUMBER(5),
  "value" NUMBER(5,2)
)

CREATE TABLE "K272G1"."PRODUCE" (
  "PersonID" VARCHAR2(10) CONSTRAINT PRODUCE_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,
  "ShowID" VARCHAR2(10) CONSTRAINT PRODUCE_ShowID REFERENCES SHOWS("showID") ON DELETE SET NULL
)

CREATE TABLE "K272G1"."PRODUCERS"(
  "PersonID" VARCHAR2(10) CONSTRAINT PRODUCERID REFERENCES PEOPLE("PersonID") ON DELETE CASCADE
)
```
```
create table "K272G1"."RATINGS" (
    "rateID" varchar2(10) NOT NULL PRIMARY KEY,
    "ratings" VARCHAR2(40)
)

create table "K272G1"."RATING_HISTORY" ( 
    "RHID" number(10,0) NOT NULL PRIMARY KEY, 
    "SHOWID" VARCHAR2(10), 
    "RATEID" VARCHAR2(10) 
)

create table "K272G1"."RECEIVE" ( 
    "showID" varchar2(10) CONSTRAINT RECEIVE_SHOWS_showID REFERENCES SHOWS("showID") ON DELETE CASCADE, 
    "rateID" varchar2(10) CONSTRAINT RECEIVE_RATINGS_rateID REFERENCES RATINGS("rateID") ON DELETE CASCADE 
)

create table "K272G1"."RECORDED_IN" ( 
    "showID" varchar2(10) CONSTRAINT RECORDED_IN_SHOWS_showID REFERENCES SHOWS("showID") ON DELETE CASCADE, 
    "cgID" varchar2(10) CONSTRAINT RECORDED_IN_CGROUPS_cgID REFERENCES COUNTRY_GROUPS("cgID"), 
    "revID" varchar2(10) CONSTRAINT RECORDED_IN_RHISTORY_revid REFERENCES REVENUE_HISTORY("revID"), 
    "curID" varchar2(10) CONSTRAINT RECORDED_IN_CURRENCIES_curID REFERENCES CURRENCIES("curID") 
)

create table "K272G1"."REVENUE_HISTORY" ( 
    "showID" varchar2(10) CONSTRAINT REVENUE_HISTORY_SHOWS_showID REFERENCES SHOWS("showID") ON DELETE CASCADE, 
    "cgID" varchar2(10) CONSTRAINT REVENUE_HISTORY_CGROUPS_cgID REFERENCES COUNTRY_GROUPS("cgID") ON DELETE CASCADE, 
    "revID" varchar2(10) NOT NULL PRIMARY KEY, 
    "amount" NUMBER(12,2), 
    "rhDate" DATE 
)

create table "K272G1"."ROLE" ( 
    "personID" varchar2(10) CONSTRAINT ROLE_personID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL, 
    "showID" varchar2(10), 
    "role" VARCHAR2(30) 
)

create table "K272G1"."SALARIES_POINTS" ( 
    "salID" varchar2(10) NOT NULL PRIMARY KEY, 
    "personID" varchar2(10) CONSTRAINT SALARIES_POINTS_personID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL, 
    "showID" varchar2(10) CONSTRAINT SALARIES_POINTS_showID REFERENCES SHOWS("showID") ON DELETE SET NULL, 
    "type varchar2(15) 
)

create table "K272G1"."SALARIES" ( 
    "salID" varchar2(10) CONSTRAINT SALARIES_SAL_PTS REFERENCES SALARIES_POINTS("salID") ON DELETE SET NULL, 
    "personID" varchar2(10) CONSTRAINT SALARIES_personID REFERENCES PEOPLE("PersonID"), 
    "showID" varchar2(10) CONSTRAINT SALARIES_showid REFERENCES SHOWS("showID"), 
    "amount" NUMBER(8,2) DEFAULT '0.00' 
)```
create table "K272G1"."SCREEN_NAMES" (
    "PersonID" varchar2(10) NOT NULL CONSTRAINT PersonID REFERENCES PEOPLE("PersonID") ON DELETE CASCADE,
    "SfName" VARCHAR2(30) NOT NULL,
    "SmName" VARCHAR2(30),
    "SlName" VARCHAR2(30) NOT NULL
)

create table "K272G1"."SHOWS" (
    "showID" varchar2(10) NOT NULL PRIMARY KEY,
    "title" VARCHAR2(30),
    "rating" varchar2(10),
    "language" VARCHAR2(15),
    "genre" VARCHAR2(15)
)

create table "K272G1"."TV_SHOWS" (
    "showID" varchar2(10) NOT NULL PRIMARY KEY CONSTRAINT TV_SHOWS_SHOWID REFERENCES SHOWS("showID") ON DELETE SET NULL,
    "startDate" DATE,
    "endDate" DATE
)

create table "K272G1"."WON_BY_PEOPLE" (
    "aiID" varchar2(10) CONSTRAINT WON_BY_PEOPLE_AWARDSINST_aiid REFERENCES AWARD_INSTANCES("aiID") ON DELETE SET NULL,
    "PersonID" varchar2(10) CONSTRAINT WON_BY_PEOPLE_PEOPLE_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL
)

create table "K272G1"."WON_BY_SHOWS" (
    "aiID" varchar2(10) CONSTRAINT WON_BY_SHOWS_AWARDSINST_aiid REFERENCES AWARD_INSTANCES("aiID") ON DELETE SET NULL,
    "showID" varchar2(10) CONSTRAINT WON_BY_SHOWS_SHOWS_showID REFERENCES SHOWS("showID") ON DELETE SET NULL
)

create table "K272G1"."WRITE" (
    "PersonID" varchar2(10) CONSTRAINT WRITE_PersonID REFERENCES PEOPLE("PersonID") ON DELETE SET NULL,
    "ShowID" varchar2(10) CONSTRAINT WRITE_ShowID REFERENCES SHOWS("showID") ON DELETE SET NULL
)

create table "K272G1"."WRITERS" (
    "PersonID" varchar2(10) CONSTRAINT WRITERID REFERENCES PEOPLE("PersonID") ON DELETE CASCADE
)

Triggers and Procedures Related to the Tables

Due to the structure of the database, we did not have to create any (direct) triggers or procedures.
Chapter 4: Data Population and Queries

Data Population

Key to the success of the POC was the initial loading (or, population) of the database. IMDB supplies a public domain version of their data, so we started with an initial load into our schema from that data source. As we did not manage to locate a public domain review source, we first did a limited “spider” sampling of a number of exiting sites in order to do a load of reviews for our database schema tests and our initial application layer. Nevertheless, this too proved very difficult and so we proceeded with building a parser for the IMDB database and a loader for our schema as explained in Chapter 7.

As difficult a task as building of the parser and the loader represented (in addition, we also had to derive the relations entirely from simple keys), we succeeded in our task to an extent as to be able to invoke/test the queries presented below.

Queries

Below, we include both the queries that we created initially for our database (see MS3) followed in each instance by the query that we implemented at the end. The results of the queries can be found in Chapter 6, Interface and Reports.

Query 1) Given time, find the top 10 box office movies in the week in North America.

```
SELECT *
FROM (SHOWS NATURAL JOIN
 (SELECT showID, amount
 FROM (SELECT *
 FROM revenue_history
 WHERE curID IN (SELECT curID
 FROM currencies
 WHERE name = 'dollar')
 AND rhDate > 'xx-xx-xxxx'
 AND rhDate < 'xx-xx-xxxx'
 AND cgID = 'abc'
 ORDER BY amount DESC)
 WHERE rownum <= 10))
;
```

The query was implemented in the database (see page 38) as shown below. In the implementation we let the user search for 5 top grossing firms while leaving the currency (for which, we have no data loaded yet) and period out. (The basic intent is to demonstrate the use of revenue histories.)

```
SELECT title, amount
FROM (SELECT *
 FROM shows s JOIN films f ON s.showid = f.showid
 JOIN revenue_history rh ON s.showid = rh.showid
 JOIN recorded_in r ON rh.showid = r.showid AND
 rh.cgid = r.cgid AND
 rh.revid = r.revid
 JOIN currencies c ON r.curid = c.curid
 WHERE c.name = 'Dollar'
 ORDER BY amount desc)
 WHERE rownum <= 5;
```
**Query 2**) Given actor, find the movies s/he acts in.

```
SELECT * 
FROM SHOWS 
WHERE showID in (SELECT showID 
  FROM act_history 
  WHERE personID in (SELECT personID 
    FROM (people NATURAL JOIN actors)
    WHERE name="John Doe")
) ;
```

The query proposed originally (see above) was later implemented in the database as shown below (see pages 44 to 45 for the results). Note that in the implementation, there are actually six queries that repeat one after the other. The first is for the movies that actor has acted in and that is the first part of the query. Basically, the query joins `act_history` to `shows` then to `salaries`, then to `salaries_points` (to grab any percentages they may receive on top of or in lieu of). We select the person of interests records (which comes from the query string). The second part is the same for the other five queries. We simply join the table of interest (in the examples case `direct` which stores all directors and the movies directed) to shows based on the person we are searching for.

```
SELECT showid, title, type, amount
FROM act_history a join shows s ON a.showid = s.showid join salaries s 
  ON a.showid = sa.showid and a.personid = sa.personid join salaries_points sp 
  ON a.showid = sp.showid and a.personid = sp.personid 
WHERE personid = :PERSONID
SELECT s.showid, s.title 
FROM direct a join shows s on a.showid = s.showid 
WHERE personid = :PERSONID
```

**Query 3**) Give the top 10 ranking movies.

```
SELECT * 
FROM (SELECT * 
  FROM (shows NATURAL JOIN films)
  ORDER BY rating DES)
WHERE rownum <= 10;
```

The query proposed originally (see above) was later implemented in the database as shown below (see page 40 for the results). The only difference is that we only showed top five movies.

```
SELECT * 
FROM (SELECT s.showid, title, avg(rateid) AS average, count(rateid) AS count 
  FROM shows s JOIN rating_history rh ON s.showid=rh.showid 
  GROUP BY s.showid, title 
  ORDER BY average DESC)
WHERE rownum <= 5;
```
Query 4) Given the title of the movie (e.g., “ABC”), find all the cast members for the movie.

```sql
SELECT *
FROM people NATURAL JOIN actors
    (SELECT personID
     FROM act_history
     WHERE showID IN (SELECT showID
                        FROM shows
                        WHERE title='The bucket list'))
);```

```
SELECT name
FROM shows s JOIN act_history a
    ON s.showid = a.showid JOIN people p
    ON a.personid = p.personid
WHERE (s.showid = :SHOWID);
```

The query proposed originally (see above) was later implemented in the database as shown below (see page 44 for the results). The query, simple at heart, was modified for the illustrative purposes as follows:

We start by joining together shows (which is where the query string begins) and `act_history` to get a table of all actors that have acted in a show. Then, we join that with `people` to get their names. Next we select only those where the `show_id` equals the query string (the one that has been selected). Then, we take only name and print that out.

Query 5) Find all the awards won by a given movie.

```sql
SELECT *
FROM (SELECT *
        FROM award_instances
        WHERE aiID IN (SELECT aiID
                        FROM won_by_shows
                        WHERE showID IN (SELECT showID
                                           FROM shows
                                           WHERE title="ABC")))
    NATURAL JOIN (SELECT * FROM awards
                    NATURAL JOIN organizations);
```

```
SELECT p.personid, p.name as name, a.name as award
FROM won_by_shows ws JOIN won_by_people wp ON
    ws.aiid=wp.aiid JOIN shows s ON
    ws.showid=s.showid JOIN people p ON
    wp.personid=p.personid JOIN award_instances ai ON
    ws.aiid=ai.aiid JOIN awards a ON
    ai.awardid = a.awardid
WHERE (s.showid = :SHOWID);
```

The query proposed originally (see above) was later implemented in the database as shown below (see page 46 for the results). Through this query, we adjusted our goal of finding all the awards won by the movie to collect all awards won by both individuals, groups, and the shows. Hence, we had to join together the `won_by_people` and `won_by_shows` table in order to a full set of awards, people, and shows. Then we joined that to `shows` and `people` to get the actual names. Then, we joined the result to `award_instances` to get a history of awards. Finally, the whole result was joined back to `awards` to get the award names. Then, we select the results that correspond to the `showid` in the query string.
Chapter 5: Triggers and Procedures

In the two sections that follow (Chapter 5.1 and Chapter 5.2), we list (and comment on) some of the triggers, procedures (and sequences) created for the site as described in the preceding chapter.

Chapter 5.1 Triggers

The two triggers listed below are for entering new records into rating_history.

**Trigger 1. Sequence Generator.**

Short description: We started with 30 since we manually entered the first set of data points. The second one is the trigger to increment the id when a new record is inserted. Similar code (not shown here) is used for People and Shows.

```sql
CREATE SEQUENCE rhSeq
    increment by 1
    start with 30
    maxvalue 9999999;
```

```sql
CREATE OR REPLACE TRIGGER increment_rhid
BEFORE INSERT ON rating_history
FOR EACH ROW
DECLARE
    v_next number;
BEGIN
    select rhSeq.nextval into v_next
    from dual;
    :new.rhid := v_next;
END;
/
```

**Trigger 2. Update of Person’s Age (when it is inserted or updated)**

Short description: First, we declared two variables, v_newAge and temp_dob. The first was used to store the newly calculated age before inserting or updating the record in the table. The second is used to store the date of birth that was just entered.

The calculation first determines how many months are between the current date (sysdate) and the new date of birth. It then takes that number and divides it by 12 to get years which is then rounded to two decimal places (this is accurate to the day). The final step is to set the new age value to the calculated age and continue the insert or update.

This trigger can also be converted into a procedure that we could run every morning. We note that some site (like IMDB) display today’s birthdays; this procedure can be used to ensure that no one is forgotten.

```sql
CREATE OR REPLACE TRIGGER update_age
BEFORE INSERT OR UPDATE OF dob ON people
FOR EACH ROW
```

```sql
BEGIN
    select rhSeq.nextval into v_next
    from dual;
    :new.rhid := v_next;
END;
/
```
Chapter 5.2  Procedures

Procedure: Country Group Display

Short Description: The following procedure displays all of the countries by groups, but only allows for the first one in the group to display the recorded date and amount. This way, we can get an accurate aggregate across multiple categories (no duplicate amounts added). This procedure simply prints out the results, but we could store them in variables and then do more with them.

```
CREATE OR REPLACE PROCEDURE revenue_country_groups(showidParam shows.showid%type)
AS
    CURSOR c1 is SELECT cgid, name, rhdate, amount
        FROM shows s JOIN revenue_history r ON
            s.showid = r.showid JOIN country_groups cg ON
                r.cgid = cg.cgid JOIN countries_make_up cm ON
                    cg.cgid = cm.cgid JOIN countries c ON cm.countryid = c.countryid
        WHERE s.showid = showidParam
        ORDER BY cgid;
    resultRow c1%rowtype;
    v_oldCGID country_groups.cgid%type;
BEGIN
    open c1;
    fetch c1 into resultRow;
    v_oldCGID := 0;
    while c1%found loop
        if(resultRow.cgid <> v_oldCGID) then
            dbms_output.put_line(resultRow.name || ' ' ||
                resultRow.rhdate || ' ' || resultRow.amount);
            v_oldCGID := resultRow.cgid;
        else
            dbms_output.put_line(resultRow.name);
        end if;
        fetch c1 into resultRow;
    end loop;
    close c1;
    commit;
END revenue_country_groups;
/
```

The procedure from the previous page returns the following result (see the next page):
set serveroutput on;
execute revenue_country_groups(1);

Canada 01-FEB-98  20000
United States
Mexico
France 02-FEB-98  30000
Spain
Israel
Italy
UK
India 03-FEB-98  748889
Iraq
Germany
Japan 04-FEB-98  40093
Korea
Chapter 6: Interface and Reports

The site for the interface is located at http://instruct.biz.uiowa.edu/courses/6K186/6K186_DatabaseAssociates/

To start off, below (Figure 1) is a screen shot of our home page. The goal of this portion was to make the interface both aesthetically pleasing and functional. The following tools were used during the construction of this site: Microsoft Visual Web Developer for the dynamic content, Adobe Dreamweaver CS3 for the page layouts, Macromedia Fireworks MX 2007 and Adobe Photoshop CS2 for the graphics (which were all handmade), Macromedia Flash MX 2007 for the image rotators (the image of Matt Damon is one of many rotating images), and FileMaker Advanced 9 for the creation of the XML file, which is where the Flash program find the image sources, as well as the JavaScript code, which powers the menu system1.

![Database Associates Internet Movie Archive](http://instruct.biz.uiowa.edu/courses/6K186/6K186_DatabaseAssociates/)

Figure 1

There are a few more additions to the site that we made in order to give it the look and feel of multimedia sites today. The first is an image popup tooltip (Figure 2 on the next page). This image shows up when you hover over any one of the preset movie posters on the page2. We also decided to

---

1 For more information and the actual code used (other than FileMaker), refer to http://instruct.biz.uiowa.edu/courses/6K070AAA/rhylock/funStuff.htm and select the SE Code tab.
2 For the full code, refer to http://instruct.biz.uiowa.edu/courses/6K070AAA/rhylock/funStuff.htm and go to the DW Example tab and select Image Rollovers and Lightbox Examples.
use the light box, more specifically LightWindow v2.0\(^3\), effect for movie trailer presentation, which is increasing in popularity. The JavaScript and CSS files are very easy to install and reference. For this project, use used references to QuickTime movies from Apple Movie Trailers\(^4\). To instantiate the light window, we just added the following parameters to the anchor tag (just after \(<a href="..."\) in the HTML code): 

\[
\text{class} = \text{"lightwindow page-options"} \quad \text{params} = \text{"lightwindow_width=320, lightwindow_height=260"}.
\]

You can see this effect in Figure 3 below.

---

\(^3\) [http://www.stickmanlabs.com/lightwindow/](http://www.stickmanlabs.com/lightwindow/)

Here is where we begin our tour. First we will cover the menu options (see Figure 4). As you can see, there are six different options to choose from on the bar and three in the header (the logo is also a link). Both Home and the logo point back to the home page. New Releases has two sub-options: In Theaters and DVD. Neither work at this point, but there is a place holder. Best Movies also has two options: Highest in US and User Rating. The first one links to Figure 5 where we can see the list of top grossing US films. This is only to demonstrate the use of revenue histories. Currently, we only needed to list the highest without aggregation because in order to combine all country group revenues into one cohesive value, we would have to take into consideration, for example, exchange and inflation rates.

The query is as follows:

```sql
SELECT title, amount
FROM (SELECT *
      FROM shows s JOIN films f ON s.showid=f.showid JOIN revenue_history rh ON s.showid=rh.showid JOIN recorded_in r ON rh.showid = r.showid AND rh.cgid = r.cgid AND rh.revid = r.revid JOIN currencies c ON r.curid=c.curid
      WHERE c.name = 'Dollar'
      ORDER BY amount desc)
WHERE rownum <= 5;
```
This query actually grabs the top 5, however, we have yet to find a reliable way to code in nested tables in ASP.NET VB (which none of us are familiar with). So, for the site, we cut it back to the inner select statement (replacing * with title, amount) for the demo. The statement itself is pretty straight forward. For the inner select, we first join together all of the required tables, then we select the currency type Dollar since we are only interested in the US. Finally we sort by amount descending. Then, we select title and amount for the join results and grab the top 5.

![Figure 4](image1.png)

The second option, User Rating, takes us to the top user ratings page for shows (Figure 6). Again, we wanted to grab only the top 5, but with the nested select statement issues, we were forced to come up with an alternative.

![Figure 5](image2.png)
The original query is as follows:

```sql
SELECT * 
FROM ( SELECT s.showid, title, avg(rateid) AS average, count(rateid) AS count 
       FROM shows s JOIN rating_history rh ON s.showid=rh.showid 
       GROUP BY s.showid, title 
       ORDER BY average desc) 
WHERE rownum <= 5;
```

We then removed the outer statement and added a constraint on how low the average rating could be. In this case we used 4.5. The modified query is below. First, we join the two tables needed, rating_history (which stores the show ID and rate ID) and shows (to get the titles). Then, we select the show id, title, average rate id (which returns the average rating), and the count of all uses that voted. This is grouped together by show id and title. Then, the found set is constrained to those at or above 4.5, and finally, sorted into descending order by average votes.

The rest of the menu pages and Contact Us are not worth writing about simply because they are place holders or have limited textual information which is irrelevant to this portion of the paper. We will now move on the search element.

```sql
SELECT s.showid, title, avg(rateid) AS average, count(rateid) AS count 
FROM shows s JOIN rating_history rh ON s.showid=rh.showid 
GROUP BY s.showid, title 
HAVING avg(rateid) >= 4.5 
ORDER BY average DESC;
```

The search box can be found in one of two places. The first is on the left-hand side on the home page (as seen in Figure 7) and the right-hand side elsewhere (as seen in Figure 8). Simply type in any portion of a person’s name or show title, and the site does the rest. Say for example, you wanted to search for anything having to do with “saving”. It could be a movie or a person (not in this case, but in some). So, we enter in “saving” (case does not matter) and we can see our results in Figure 9.
The result set include Movies, Actors, Directors, Producers, Composers, Editors, and Writers. You simply select the tab and the results are posted. The query for this is really simple. We return the entire set of values (which are later scaled back on the site) from the shows (this is tab Movies) where the search term NAME (which is parsed from the query string) is in the title. We convert everything to lowercase in order to avoid any case-sensitive issues.

```sql
SELECT *
FROM shows
WHERE (lower(title) LIKE '%' || lower(:NAME) || '%');
```

After selecting a tab and object, you are taken to the details view for that object. The tour will continue using “Saving Private Ryan”. The details for this movie are below in Figure 10. As you can see, the basics about the movie are listed. In the full version, we of course would have all of the possible information we could collect, but for this demo, we simply added a few items. Like the search results, each show has a tab set as well. This is to keep the page from going on forever like we have all seen on other site. By keeping the information tightly packed and organized, we hope to increase the ease of which people browse for movie related information.

In Figure 10 (see the next page), there are two new categories: Awards and Revenues. Under Awards, it lists all awards associated with the movie, both individual, group, and by show. Under Revenues, we have a simple view of the amounts received so far by country group. Again, this is just for demo purposes so we simply listed the values instead of actually putting them into context such as the value at the date recorded and how comparable it is to today (inflation).
The country code query is as follows:

```
SELECT cgid, rhdate, amount
FROM shows s JOIN revenue_history r ON
  s.showid = r.showid JOIN country_groups cg ON
  r.cgid = cg.cgid JOIN countries_make_up cm ON
  cg.cgid = cm.cgid JOIN countries c ON
  cm.countryid = c.countryid
WHERE (s.showid = :SHOWID)
GROUP BY cgid, rhdate, amount
ORDER BY rhdate;
```

Basically, countries are clustered together by for revenue reporting. As you can see in Figure 10, country group 1 consists of the United States, Canada, and Mexico. We just put these together to show how it works. In reality, they would be clustered by currency values. The query groups by the group id, recorded date, and amount and returns only all unique group results. We also have a procedure for listing all of the countries and for the first in each group, we list the date and amount. That procedure can be found in Chapter 5.2 (Procedures) on pages 35 and 36 (output).

![Figure 10](image)

Now, we will perform a new search for “morgan”. This will bring up actor Morgan Freeman. Select him (the results are in Figure 11). As you can see, we have a comprehensive list of all the categories rolled into one. This could be broken up into tabs like the others, but we will have to wait and see if this necessary.

The query is as follows:

```
SELECT showid, title, type, amount
FROM act_history a join shows s
  ON a.showid = s.showid join salaries sa
  ON a.showid = sa.showid and a.personid = sa.personid join salaries_points sp
  ON a.showid = sp.showid and a.personid = sp.personid
WHERE personid = :PERSONID

SELECT s.showid, s.title
FROM direct a join shows s ON a.showid = s.showid
```
WHERE personid = :PERSONID

There are actually six queries that repeat one after the other. The first is for the movies that actor has acted in and that is the first part of the query above. Basically, the query joins act_history to shows then to salaries, then to salaries_points (to grab any percentages they may receive on top of or in lieu of). We select the person of interests records (which comes from the query string). The second part is the same for the other five queries. We simply join the table of interest (in the examples case direct which stores all directors and the movies directed) to shows based on the person we are searching for.

From here, select “The Bucket List” (Figure 12). As you can see, there are two actors, Morgan Freeman and Jack Nicolson listed.

The query to return these actors is as follows:

```sql
SELECT name
FROM shows s JOIN act_history a ON
    s.showid = a.showid JOIN people p ON
    a.personid = p.personid
WHERE (s.showid = :SHOWID);
```

The query for this is very simple. We start by joining together shows (which is where the query string begins) and act_history to get a table of all actors that have acted in a show. Then, we join that with people to get their names. Next we select only those where the show id equals the query string (the one that has been selected). Then, we take only name and print that out.
Next, select the awards tab. As you can see (Figure 13), Jack Nicholson won an Oscar for his performance. The query to return this value is as follows:

```
SELECT p.personid, p.name as name, a.name as award
FROM won_by_shows ws JOIN won_by_people wp ON
    ws.aiid=wp.aiid JOIN shows s ON
    ws.showid=s.showid JOIN people p ON
    wp.personid=p.personid JOIN award_instances ai ON
    ws.aiid=ai.aiid JOIN awards a ON
    ai.awardid = a.awardid
WHERE (s.showid = :SHOWID);
```

Here, our goal is to collect all awards won by both individuals, groups, and the shows as mentioned earlier. So, we have to join together the `won_by_people` and `won_by_shows` table in order to a full set of awards, people, and shows. Then we join that to shows and people to get the actual names. Then, it is joined to `award_instances` to get a history of awards. Finally, it is joined back to awards to get the award names. They, we select the results that correspond to the show id in the query string.

Now we will move on the inserts. First, go back to the home page and scroll down to the bottom and select the Admin button (Figure 14). This will bring up a list of items to edit (Figure 15). There are only two here, the other is rating a movie which I will discuss later. Both look and feel the same, so we are going to cover Shows. So, select Shows and we will get started.
In Figure 16 above, we see the view for the insert, update, and delete process. In the screen shot, I have selected Air Force One to populate the details view to the right. Here, you can select either to edit or delete this record, or add a new one.

Finally, this site has the ability to save user ratings of a particular movie. To get to this page, simply select Rate A Show on any page from the links panel on either the right or left side. Once there, click New to begin the process (this brings you to Figure 17). To rate a show, select the title from the list box and then select a number of stars from the drop down list (5 being the best). Once you do, hit Insert and you will have something that looks like Figure 18 which confirms your rating.
The code for this was written using Visual Basic and is listed below. Basically, it performs the insert into the database for the specified data source, then retrieves the variables from the list box and drop down list, and then passes those values to the review page (Figure 18) via a query string which is then parsed by the page.

```vbnet
<%<script runat="server">  Protected Sub InsertButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) FormView1.InsertItem(True) Dim newshowid = CType(FormView1.FindControl("ListBox1"), ListBox).SelectedItem.Value Dim newrating = CType(FormView1.FindControl("DropDownList1"), DropDownList).SelectedValue Dim newurl = "Http://instruct.biz.uiowa.edu/courses/6k186/6k186_databaseassociates/editRatingsSubmit.aspx?showid=" + newshowid + "&rating=" + newrating + "&btnSubmit" Response.Redirect(newurl, True) End Sub </script>%
```

There are many different types of functionality and security that we will add to this site in the future. We are still far from complete when it comes to calculating currency values across time for the purpose of comparison. Also, we need to add more insert/update/delete form for all topics. This is where the security comes in. We would like to have the content generated much in the same was as a Wiki page does with open editing, but we do not want it to be entirely open. We will have to come up with some sort of validation process submissions or allow only those users with relevant backgrounds and who have proven themselves to be trustworthy. This is still open to debate, but these are some of the ideas. Also, the editing can be found by clicking on Admin on the home page. This of course will be removed and replaced with a login control.
Chapter 7: Conclusions and Implementation Plan

What we learned
Adapting data that already exists within a functional framework can be difficult, especially if the application that you are creating doesn’t share all of the relations and functional aspects in an easily mappable manner. This project was not difficult to design, but building a parser for the IMDB database and a loader for our schema was more difficult than originally expected. The database, while public domain, is in a text format, and the relations have to be derived entirely from simple keys. However, due to the nature of our theme (movies), the IMDB database was the richest target for an initial source of much of our data.

Implementation
The implementation of this system should be straight forward. We didn’t make use of any Oracle-specific constraints, so a system utilizing PostgreSQL or MySQL are both acceptable DB platforms to start with.

An implementation plan should include:

1. Selection of the database engine.
2. Selection of the server topology for the web-site. The server and the database could easily run on the same machine, but as user load increases it might be desirable to run the web-server and the database server on separate machines.
3. Identification and contract with web-hosting service. The options here are varied. Option 1 would include absorbing the costs for a server topology. Option 2 would utilize existing vendors for hosting. There are several hosting providers- in fact both Amazon.com and Yahoo.com have established business accounts at a very reasonable rate. Amazon offers full MySQL support, and Yahoo offers a full hosting service for very reasonable rates. Since this is an entirely Web-based application, the costs for off-site hosting offer an attractive alternative to purchasing and maintaining a development and production server.
4. Create parser for IMDB files. These are downloadable through a link on the IMDB web-site. The files need to be cleaned, and parsed into CVS format.
5. Loading of data from CVS files into database.
6. Register Web Domain.
7. Select Web Hosting option.
8. Creation of Web interface.
9. Design sign-off and Interface Testing,
10.  Web site QA testing.

Please, see the following two pages for the Contract Estimate Summary – Option 1 (p. 49) and for the Contract Estimate Summary – Option 1 (p. 50).
## 7.A Appendix

### Contract Estimate Summary – Option 1

<table>
<thead>
<tr>
<th>Option 1 - Self Hosting</th>
<th>Rate</th>
<th>Hours</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-Programmer</td>
<td>$100</td>
<td>12</td>
<td>$1,200</td>
</tr>
<tr>
<td>DBA</td>
<td>$125</td>
<td>20</td>
<td>$2,500</td>
</tr>
<tr>
<td>Design</td>
<td>$75</td>
<td>10</td>
<td>$750</td>
</tr>
</tbody>
</table>

1 Year support software support and monthly change contract (Note: The support includes one stance of software maintenance/upgrades as well as two (2) hours of Web site updates/changes per month) 

<table>
<thead>
<tr>
<th></th>
<th>$3,000</th>
<th>NA</th>
<th>$3,000 Yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Server</td>
<td>$2,000</td>
<td>NA</td>
<td>$2,000</td>
</tr>
<tr>
<td>Web Server</td>
<td>$2,000</td>
<td>NA</td>
<td>$2,000</td>
</tr>
</tbody>
</table>

Yearly Hardware Support Contract (includes 24x7 1 hour downtime response) 

<table>
<thead>
<tr>
<th></th>
<th>$1,000</th>
<th>NA</th>
<th>$1000 yearly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yearly Domain registration</td>
<td>$20</td>
<td>NA</td>
<td>$20 yearly</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start-up costs</th>
<th>$12,470</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annual costs</td>
<td>$4,020</td>
</tr>
</tbody>
</table>
## Contract Estimate Summary – Option 2

<table>
<thead>
<tr>
<th>Option 2 - Remote Hosting</th>
<th>Rate</th>
<th>Hours</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-Programmer</td>
<td>$100</td>
<td>12</td>
<td>$1,200</td>
</tr>
<tr>
<td>DBA</td>
<td>$125</td>
<td>20</td>
<td>$2,500</td>
</tr>
<tr>
<td>Design</td>
<td>$75</td>
<td>10</td>
<td>$750</td>
</tr>
<tr>
<td>1 Year support software support and monthly change contract (Note: The support includes one stance of software maintenance/upgrades as well as two (2) hours of Web site updates/changes per month)</td>
<td>$3,000</td>
<td>NA</td>
<td>$3,000 Yearly</td>
</tr>
<tr>
<td>Yearly Domain registration</td>
<td>$20</td>
<td></td>
<td>$20 yearly</td>
</tr>
<tr>
<td>Yearly Hosting Fee(Yahoo)</td>
<td>$20</td>
<td></td>
<td>$20 yearly</td>
</tr>
<tr>
<td>Start-up costs</td>
<td></td>
<td></td>
<td>$6740</td>
</tr>
<tr>
<td>Annual costs</td>
<td></td>
<td></td>
<td>$3,040</td>
</tr>
</tbody>
</table>