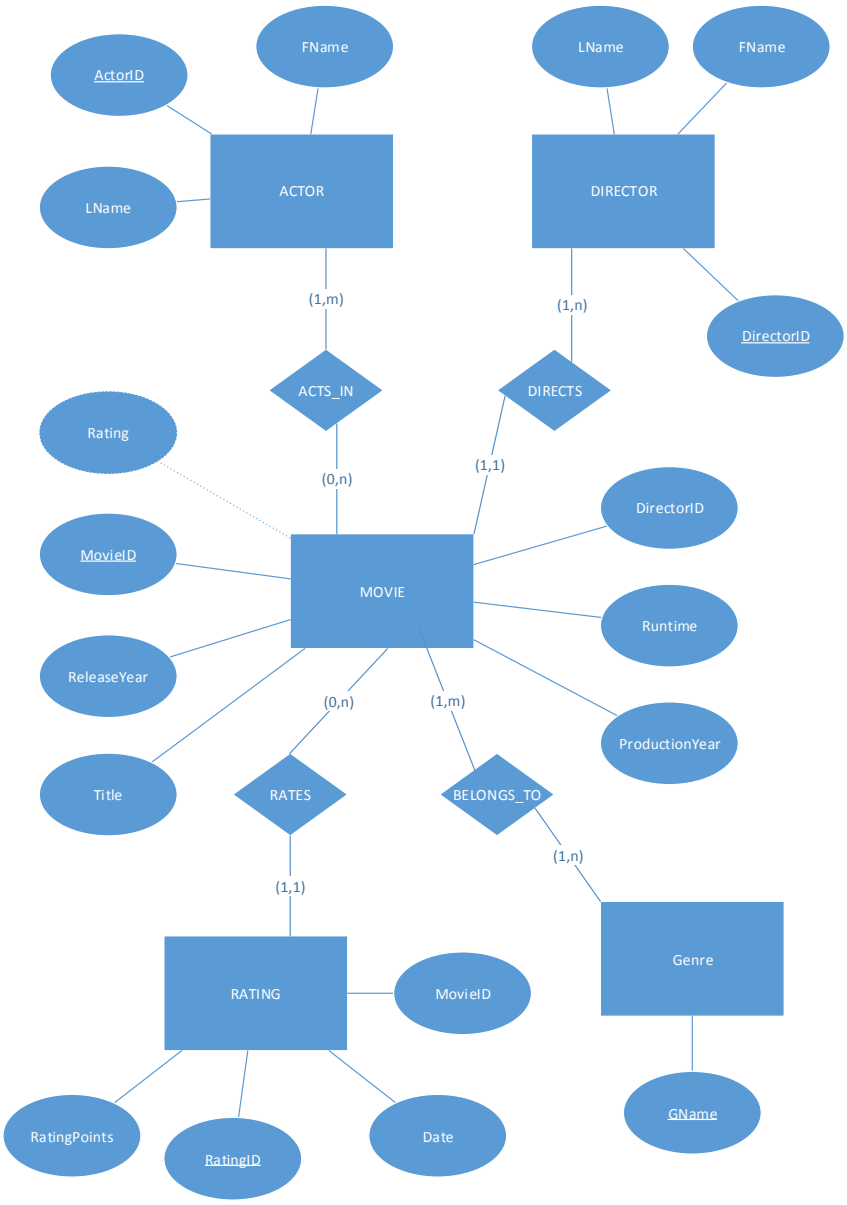
Dokumentation zum Datenbankprojekt

Katharina Chowanski / Marvin Kleinert / Marius Schidlack

22. Juli 2015

Iteration 1: Modellierung

1. Entity-Relationship-Modell



¡¡¡¡¡¡¡ HEAD

Relationales Modell

MOVIE (Title: varchar(255); MovieID: int8; ReleaseYear: interval YEAR; ProductionYear: interval YEAR; Runtime: int4; DirectorID: int8; Rating: float4)

ACTOR (ActorID: int8; FName: varchar(255); Lname: varchar(255))

DIRECTOR (DirectorID: int8; FName: varchar(255); LName: varchar(255))

RATING (RatingID: int8; Date: date; RatingPoints: int4; MovieID: int8)

GENRE (GName: varchar(255))

ACTS_IN (MovieID: int8; ActorID: int8)

BELONGS_TO (GName: varchar(255); MovieID: int8)

=======

2. Relationales Modell

MOVIE (Title: string; MovieID: int; ReleaseYear: int; ProductionYear: int;
Runtime: int; DirectorID: int; Rating: float)

ACTOR (ActorID: int; FName: string; Lname: string)

DIRECTOR (DirectorID: int; FName: string; LName: string)

RATING (RatingID: int; Date: date; RatingPoints: int; MovieID: int)

GENRE (GName: string)

ACTS_IN (MovieID: int; ActorID: int)

BELONGS_TO (GName: string; MovieID: int) ¿¿¿¿¿¿ f38e48f7e0f459601f6e746e08ba378863513dba

3. CREATE-Statements

```
CREATE TABLE MOVIE
(Title              varchar(255) NOT NULL,
MovieID             int8 NOT NULL,
ReleaseYear         interval YEAR,
ProductionYear          interval YEAR,
Runtime             int4 ,
DirectorID          int8 NOT NULL,
Rating              float4 ,
PRIMARY KEY (MovieID),
FOREIGN KEY (DirectorID) REFERENCES DIRECTOR (DirectorID)
);

CREATE TABLE ACTOR
(ActorID            int8 NOT NULL,
FName               varchar(255),
```

```
LName                varchar(255),
PRIMARY KEY (ActorID)
);

CREATE TABLE DIRECTOR
(DirectorID          int8 NOT NULL,
FName                varchar(255),
LName                varchar(255),
PRIMARY KEY (DirectorID)
);

CREATE TABLE RATING
(RatingID            int8 NOT NULL,
Date                 date,
RatingPoints             int4,
MovieID              int8 NOT NULL,
PRIMARY KEY (RatingID),
FOREIGN KEY (MovieID) REFERENCES MOVIE (MovieID)
);

CREATE TABLE GENRE
(GName               varchar(255) NOT NULL,
PRIMARY KEY (GName)
);

CREATE TABLE ACTS_IN
(MovieID             int8 NOT NULL,
ActorID              int8 NOT NULL,
PRIMARY KEY (MovieID, ActorID),
FOREIGN KEY (MovieID) REFERENCES MOVIE(MovieID),
FOREIGN KEY (ActorID) REFERENCES ACTOR (ActorID)
);

CREATE TABLE BELONGS_TO
(GName               varchar(255) NOT NULL,
MovieID              int8 NOT NULL,
PRIMARY KEY (GName, MovieID),
FOREIGN KEY (GName) REFERENCES GENRE (GName),
FOREIGN KEY (MovieID) REFERENCES MOVIE (MovieID)
);
```

Iteration 2: Datentransformation und API

1. Veränderungen im Relationalen Modell

MOVIE (Title: text; MovieID: varchar(9); ReleaseYear: int2; ProductionYear: int2; Runtime: int2; DirectorID: int; Rating: numeric(3,1), RatingCount: int)

ACTOR (ActorID: varchar(9); FName: varchar(127); LName: varchar(127))

DIRECTOR (DirectorID: varchar(9); FName: varchar(127); LName: varchar(127))

RATING (RatingID: varchar(9); Date: date; RatingPoints: int2; MovieID: varchar(9))

GENRE (GName: varchar(127))

ACTS_IN (MovieID: varchar(9);
ACTOR.FName: varchar(127); ACTOR.LName: varchar(127))

DIRECTS (MovieID: varchar(9); DIRECTOR.FName: varchar(127); DIRECTOR.LName: varchar(127))

BELONGS_TO (GName: varchar(127); MovieID: varchar(9))

Die Änderungen zum ursprünglichen Relationalen Modell resultieren aus der Sichtung des Datensatzes und bestehen im Wechsel der Primärschlüssel bei den Relationen ACTOR und DIRECTOR von einer in den Daten nicht enthaltenen ID zu Vor- und Nachname, sowie einer neuen Relation DIRECTS. Diese ist nötig geworden, da in den Daten auch Filme vorkommen, die mehrere Directors haben können und somit zwischen MOVIE und DIRECTOR eine n:m-Beziehung besteht. Die Verwendung von Vor- und Nachnamen der Actor und Director als Primärschlüssel vereinfachte zudem die Implementierung.

2. Datentransformation mit Java und JDBC

Bei Betrachtung der CSV Datei fällt auf, dass jede Zeile in der Datei jeweils einen Datenbank Eintrag repräsentiert. Zur Datentranformation in Java haben wurde daher ein Crawler Programmiert, der die Datei Zeile für Zeile durchgeht:

```
BufferedReader reader = new BufferedReader( new FileReader (csv_file));
String         line = null;

...

try {
    for (int i=0;(line = reader.readLine() ) != null;i++) {
        ...
    }
    ...
    reader.close();
} catch (Exception e) {
    System.err.println( e.getClass().getName()+": "+ e.getMessage() );
}
```

Auf die einzelne Zeile lässt sich dann bei jedem Schleifendurchlauf jeweils durch die Variable "lineßugreifen. Die einzelnen Attribute sind in der Zeile jeweils durch Kommata getrennt. Mit der Java Methode ßplitïst es sehr einfach möglich derartige Aufzählungs-Strings in seine Einzelteile zu zerlegen und in einem Array aus Strings zu speichern:

```
String[] attr = line.split("\t");
```

Nach einigen Testläufen fiel auf, dass vereinzelt Einträge wie NA oder andere Unregelmäßigkeiten bei den Attributen auftreten. Diese Fehler mussten nun mithilfe der Eigentlichen Datentransformation behoben werden. Die ID wird einfach übernommen, da sie bei allen Einträgen einheitlich ist. Beim title gab es einige Apostrophe, die aber in SQL eine semantische Bedeutung haben. Ein Apostroph wird daher in SQL einfach zweimal geschrieben um es zu escapen. Für das rel_year, das rating und die duration müssen jeweils NA oder anderen ungültige Zahlen erkannt und durch NULL ersetzt werden. Dazu dient die Hilfsmethode parseIntIfPossible()

```
String id = attr[0];
String title = attr[1].replaceAll("'", "''");;
String rel_year = this.parseIntIfPossible(attr[2].substring(0, 4));
String rating = this.parseFloatIfPossible(attr[3]);
String rating_count = this.parseIntIfPossible(attr[4]);
String duration = attr[5].indexOf(' ')>=0?attr[5].substring(0,attr[5].indexOf(
duration = this.parseIntIfPossible(duration);
```

Die Actors und Genres wurden bei jedem Film jeweils als Multi-Value Attribut angegeben, wobei die einzelnen Einträge jeweils durch "getrennt waren. Auch hier wurden diese wieder zu einem String Array gesplittet, dann die Apostrophe ersetzt. Anschließend mussten die einzelnen Namen noch einmal in Vor- und Nachname gesplittet werden.
Die Einzelnen Filme, Director, Genres und Actors konnten nun mit INSERT Statements in die Datenbank eingefügt werden. Bei Actor, Director und Genre musste außerdem noch unterschieden werden, ob der Eintrag bereits existiert, um Datenredundanz zu vermeiden.

```
String sql = "INSERT INTO movies (id,title,rel_year,rating,rating_count,duratio
"('"+id+"','"+title+"',"+rel_year+","+rating+","+rating_count+","+duration+",'
...
String sql =
"INSERT INTO directors"+
"(firstname, lastname)"+
" SELECT '"+name[0]+"', '"+name[1]+"'"+
"WHERE NOT EXISTS ("+
" SELECT firstname,lastname FROM directors WHERE firstname = \'"+name[0]+"\' A
" ); ";
...

sql += "INSERT INTO actors"+
"(firstname, lastname)"+
" SELECT '"+name[0]+"', '"+name[1]+"'"+
"WHERE NOT EXISTS ("+
" SELECT firstname,lastname FROM actors WHERE firstname = '"+name[0]+"' AND las
" ); ";

sql += "INSERT INTO acts_in"+
"(firstname,lastname,movieID)"+
" SELECT '"+name[0]+"','"+name[1]+"','"+movieID+"'"+
"WHERE NOT EXISTS ("+
" SELECT firstname,lastname,movieID FROM acts_in WHERE firstname = '"+name[0]+
...

sql += "INSERT INTO genres"+
"(name)"+
" SELECT '"+genre+"'"+
"WHERE NOT EXISTS ("+
" SELECT name FROM genres WHERE name = '"+genre+"'"+
" ); ";

sql += "INSERT INTO has_genre"+
"(movieID,genre)"+
" SELECT '"+movieID+"', '"+genre+"'"+
"WHERE NOT EXISTS ("+
" SELECT movieID,genre FROM has_genre WHERE movieID = '"+movieID+"' AND genre
...
```

Ein SQL Statement kann dann wie folgt ausgeführt werden:

```
Statement stmt = c.createStatement();
...
stmt.execute(sql);
```

3. Datentransformation mit SQL

```
CREATE TABLE alldata
    (imdbID          VARCHAR(9),
    name             TEXT,
    year             INT2,
    rating           NUMERIC(3,1),
    votes            INT,
    runtime          INT2,
    directors        TEXT,
    actors           TEXT,
    genres           TEXT);

COPY alldata
FROM 'imdb_top100t_2015-06-18.csv'
FORMAT CSV
DELIMITER '\t';

INSERT INTO movies
(imdbID, title, rel_year, rating, rating_counts, duration)
SELECT imdbID, name, year, rating, votes, runtime
FROM alldata A
WHERE NOT EXSISTS   (SELECT imdbID
                     FROM movies
                     WHERE A.imdbID = imdbID);
```

Unsere Idee bei der Transformation der Daten ausschließlich mit SQL bestand darin, zunächst eine Tabelle für alle Attribute anzulegen, um dann mit der von SQL zur Verfügung gestellten COPY-Funktion die CSV-Datei mit ihrem Delimiter TAB in die Tabelle zu kopieren. Die gewünschten Daten für die einzelnen Tabellen unserer Datenbank erhält man schließlich durch Querys auf die neu erzeugte Tabelle.

4. API

```
public interface DatabaseAPI {
    public String bestMovies (int count);
    public String moviesRating (int from, int to);
    public String moviesRating (int from, int to, int year);
    public String actorsInMovie (String movieTitle);
    public String directorsOfMovie (String movieTitle);
    public String ratingOfMovie (String movieTitle);
    public String genresOfMovie (String movieTitle);
    public String movieInfos (String movieTitle);
    public String moviesOfActor (String fname, String lName);
    public String moviesOfActor
                    (String fname, String lName, int year);
    public String debutOfActor (String fName, String lName);
```

```
    public String  moviesOfDirector
                    (String fName, String lName);
    public String  moviesOfDirector
                    (String fName, String lName, int year);
    public String actorsWorkedForDirector
                    (String fName, String lName);
    public String directorWithMostMovies (int year);
```

Unsere API orientiert sich an den Beispielanfragen in der Projektbeschreibung, umfasst aber zudem noch weitere nützliche Anfragen, wie z.B. welche Schauspieler in einem bestimmten Film mitspielen oder wer bei einem bestimmten Film Regie geführt hat. Exemplarisch folgt nun die Implementierung von zwei Methoden des obigen Interfaces:

```
public String actorsInMovie (String movieTitle){
    String sql =    ("SELECT firstname, lastname "
                    + "FROM movies, acts_in "
                    + "WHERE title = ? "
                    + "AND id = movieID ");
    TupleQ[] pstValues = {new TupleQ(1,movieTitle)};
    ResultSet rs = manageQuery (sql, pstValues);
    String retString = resultTable(rs, new TupleR[]
            {new TupleR("firstname",String.class),
            new TupleR( "lastname", String.class) });
    return retString;
}

public String debutOfActor (String fName, String lName){
    String sql =    ("SELECT title, rel_year "
                    + "FROM movies, acts_in "
                    + "WHERE firstname = ? "
                    + "AND lastname = ? "
                    + "AND id = movieID "
                    + "AND rel_year > 0 "
                    + "ORDER BY rel_year ASC "
                    + "LIMIT 1 ");
    TupleQ[] pstValues = {new TupleQ(1,fName),
                          new TupleQ(2, lName)};
    ResultSet rs = manageQuery (sql, pstValues);
    String retString = resultTable(rs, new TupleR[]
            {new TupleR("title",String.class),
                new TupleR("rel_year",int.class)});
    return retString;
}
```

Hauptbestandteil der Methoden bildet die SQL-Abfrage, die zusammen mit den nötigen Parametern in einer separaten Funktion weiterverarbeitet wird, um sich wiederholenden Code zu vermeiden. Dabei werden PreparedStatements verwendet, um mögliche SQL-Injections durch falsche Eingaben zu verhindern. Auch die Ausgabe der Daten wird in einer separaten Funktion gemanagt, sodass mögliche Änderungen in der Darstellung für die GUI leicht an nur einer Stelle vollzogen werden können.

Iteration 3: GUI und Datamining

1. Datamining

Auswertung des Projekts