Projektaufgabe Filmdatenbank

Teil der Prüfungsleistung im Modul

Datenbanken 1 (EDV-Nr. 113210)

Sommersemester 2019

Projektgruppe **08** mit den Gruppenteilnehmern:

Heinrich, Leon	37560	MI-7
Dubiel, Sebastian	37438	MI-7
Vu, Hai	37480	MI-7

Hiermit bestätigen die oben genannten Gruppenteilnehmer, dass die Projektaufgabe ohne Unterstützung dritter und unter Nennung aller Quellen und eingesetzten Hilfsmittel erstellt wurde. Es ist uns bekannt, dass ein Verstoß gem. § 17 Abs. 5 Satz 1 zum Nichtbestehen der kompletten Prüfungsleistung im Modul Datenbanken 1 (EDV-Nr. 113210) führt.

Stuttgart, den 17. Juli 2019

Heinrich, Leon

Dubiel, Sebastian

Vu, Hai

SYSTEMBESCHREIBUNG

Tabellen

Unsere Filmdatenbank haben wir mit insgesamt 9 Tabellen aufgebaut. Die zentrale Tabelle stellt dabei die Tabelle `movies` dar:
Diese beinhaltet den künstlichen Primärschlüssel `movie_id`, sowie die Nicht-Schlüsselattribute `title` (Film-Titel), `plot_outline` (Film-Zusammenfassung) und `release_date` (Veröffentlichungsdatum). Des weiteren enthält sie die Fremdschlüssel `m_comp_id` und `m_dir_id`, welche die (N:1)-Beziehungen zwischen der Tabelle `movies` und den Tabellen `prodCompany` bzw. `director` realisieren. Dies begründen wir damit, dass ein Film von genau einer Filmproduktionsgesellschaft produziert wird und genau einen Regisseur hat, jedoch eine Filmproduktionsgesellschaft/ein Regisseur einen oder mehrere Filme produzieren/inszeniert.

Die Tabelle `director` beschreibt dabei die Eigenschaften des Regisseurs durch `dir_socSecNum` (Sozialversicherungsnummer), `dir_name` (Name) und `dir_birthday` (Geburtsdatum). Die Sozialversicherungsnummer stellt den Primärschlüssel dar, welcher in diesem Fall, ein nicht-künstlicher Schlüssel ist.

Die andere von `movies` referenzierte Tabelle `prodCompany` umfasst wiederum alle Filmproduktionsgesellschaften und enthält neben der eindeutigen ID (`comp_id`) Informationen über den Namen der Filmproduktionsgesellschaft `(comp_name)` und den Fremdschlüssel `p_address_id`, welcher wiederum eine PK/FK-Beziehung (N:1) zur Tabelle `address` ermöglicht. Wir haben angenommen, dass sich evtl. mehrere Filmproduktionsgesellschaften/Tochterfirmen an einem Ort angesiedelt haben und somit eine Adresse zu mehreren Filmproduktionsgesellschaften zugeordnet werden kann.

In der Tabelle `address` sind dann genauere Informationen über die Adresse der Filmproduktionsgesellschaft/en festgehalten. Dazu gehören die Postleitzahl (`postCode`), Straße (`street`), Stadt/Ort (`city`), Land (`country`) und der Primärschlüssel `address_id`.

Hinzu kommt eine Tabelle `movieRole`. Diese beinhaltet Daten zur jeweiligen Rolle in einem Film. So sind hier neben der ID `role_id` auch der Name der Rolle (`role_name`) eingetragen. Zusätzlich wurden hier zwei Fremdschlüssel `mR_actor_id` und `mR_movie_id` definiert. `mR_movie_id` bezieht sich dabei auf die Tabelle `movies` (N:1) und `mR_actor_id` stellt eine Referenz zur Tabelle `actor` dar (N:1). Diese Beziehungen beruhen darauf, dass eine Rolle zu genau einem Film gehört (Filmfortsetzungen/mehrteilige Filme haben wir ausgeschlossen), jedoch in einem Film mehreren Rollen vorkommen. Andererseits wird eine

Rolle nur von einem Schauspieler belegt (Remakes ausgeschlossen), allerdings kann ein Schauspieler mehrere Rollen annehmen.

Die Daten aller beteiligten Schauspieler sind in der Tabelle `actor` gespeichert, wozu `actor_socSecNum` (Sozialversicherungsnummer), `actor_name` (Name) und `actor_birthday` (Geburtstag) zählen. `actor_socSecNum` ist hier der Primärschlüssel und ist somit zusammen mit `dir_socSecNum` der einzige nicht-künstliche Schlüssel in unserem Projekt.

Als nächstes ist die Tabelle `mov_gen` zu nennen. Diese setzt die einzige "logische" N:M-Beziehung in unserem Projekt in zwei 1:N-Beziehungen um. So sind neben der ID `mov_gen_id`, eine Referenz auf die Tabelle `movies (mg_movie_id)`, sowie auf die Tabelle `genre (mg_genre_id)` gegeben. Durch `mov_gen` konnte die Tatsache umgesetzt werden, dass ein Film mehrere Genres haben kann, aber auch ein Genre mehreren Filmen zugeordnet wird. `mg_movie_id` und `mg_genre_id` haben wir zudem kombiniert als `UNIQUE` definiert, um redundante Genre-Zuweisungen zum gleichen Film zu verhindern.

Der referenzierte Primärschlüssel `genre_id` und der Name des Genres (`genre_name`) bilden dabei die Tabelle `genre`.

Unsere letzte Tabelle `movie_grosses` ist für das Speichern der Einnahmen eines Filmes vorgesehen, wobei wir uns nur auf Kinoeinnahmen (`movie_theater`) beschränkt haben. Des weiteren wird das zu den Einnahmen zugehörige Datum (`grossDate`) und eine ID (`grosses_id`) gespeichert. Zudem enthält sie auch den benötigten Fremdschlüssel (`g_movie_id`) um die Zuordnung zur Tabelle `movie` zu ermöglichen. Folglich kann ein Film mehrere Einnahmen generieren (N:1-Beziehung).

Datentypen

In unseren Tabellen haben wir alle geforderten Datentypen verwendet:
Einerseits haben wir zum Anlegen von IDs *ganze Zahlen* (`INTEGER`) verwendet, welche numerisch exakt sein sollen.
Andererseits haben wir den Datentyp `NUMBER` gewählt, da man durch die Angabe von zwei *Nachkommastellen* sehr gut Geldbeträge, in unserem Fall für die Kinoeinnahmen (`movie_theater`) darstellen kann.
Für *Zeichenketten mit variabler Länge* (vor allem Namen) haben wir den Datentyp `VARCHAR(x)`, mit einer für den Zweck jeweils ausreichenden Länge gewählt.
Bei *Zeichenketten* mit kurzer und *fester Länge* haben wir uns für den Datentyp `CHAR(x)` entschieden. Dazu zählen bei uns Sozialversicherungsnummern oder Postleitzahlen.

Um ein *Datum* in einer Tabelle zu implementieren, wie Geburtsdaten oder die Daten der jeweiligen Kinoverkaufszahlen (`grossDate`) haben wir standardmäßig den Datentyp `DATE` verwendet.

Aufbau SQL Script

Um ein völlig restart-fähiges Skript zu garantieren beginnen wir als erstes mit dem Löschen (`DROP`) aller `CONSTRAINTs`. Zuerst werden hierbei alle `FOREIGN KEYs`, die `UNIQUE CONSTRAINTs` und erst dann alle `PRIMARY KEYs` gedroppt. Diese Reihenfolge mussten wir wählen, damit sich beim Entfernen der PKs keine FKs mehr auf diese beziehen.
Nach dem Löschen aller `CONSTRAINTs` können alle Tabellen entfernt werden.
Danach werden zunächst alle Tabellen ohne ihre PK/FK-`CONSTRAINTs` erstellt, jedoch mit den zugehörigen Datentypen und Column-`CONSTRAINTS` `NULL / NOT NULL`. Erst im Anschluss erhalten sie dann die jeweils nötigen Table-`CONSTRAINTs`. Dies dient wiederum der Restart-Fähigkeit des Skriptes.
Dabei verteilen wir die `CONSTRAINTs` in umgekehrter Reihenfolge zum Löschen, also erst die PKs, dann `UNIQUE` und zuletzt die FKs.
Im Anschluss können die Tabellen dann befüllt werden (`INSERT`), wobei wir mindestens vier `INSERTs` pro Tabelle getätigt haben, teilweise auch mehr, um 1:N-Beziehungen zu verdeutlichen und mehrere Einträge mit gleichem FK zu speichern.
Vor einigen `INSERTs` sind `SEQUENCEs` und `TRIGGER` wiederzufinden. Hierbei gibt es zwei Arten von `TRIGGER`.
Die erste Art von `TRIGGER` (`<..>_incrementId`) ist dafür verantwortlich, IDs stets um 1 zu inkrementieren. Dies ist dafür nützlich, um viele Einträge automatisch mit gleichmäßig aufsteigenden und vor allem eindeutigen Werten zu versehen. Dazu haben wir zunächst alle IDs der Einträge in Tabellen, welche den `TRIGGER` verwenden, auf 0 gesetzt. Beim Ausführen des Programms werden diese dann durch den `TRIGGER` angepasst.
Der andere `TRIGGER` (`prevent_future_grosses`) überprüft vor dem Einfügen/Ändern von Einträgen in die/der Tabelle `movie_grosses` das Veröffentlichungsdatum des zugehörigen Filmes. Falls das Datum noch nicht gesetzt wurde (`NULL` ist) oder das Datum in der Zukunft liegt, jedoch Einnahmen zugewiesen werden (`movie_theater`), wird eine eigens erstellte Exception geworfen, die das Einfügen verhindert. In der Realität könnten dadurch Fehler bei der Interpretation der Einspielergebnisse verhindert werden.

Extra SQL-Skript

Getrennt vom restlichen SQL-Skript, haben wir unsere `SELECT/UPDATE`-Befehle, sowie die `TRIGGER`-Tests gespeichert

Select-Befehle

Hier haben wir neben dem Ausgeben aller Daten einer Tabelle, einfache Joins, Vergleichsoperationen, `LEFT OUTER JOINS`, als auch `SUM/COUNT`-Funktionen, kombiniert mit `GROUP BY` (Gruppierung) und `DESC` (absteigende Sortierung), verwendet.

Update-Befehle

Hier wurden einzelne Attributwerte ersetzt (u.a. auch Datumswerte), vertauscht oder auch durch Addition erhöht.

Testen von Triggern

Die Increment-`TRIGGER` haben wir durch `SELECT`-Abfragen getestet, welche aufeinanderfolgende Entitäten mit der geforderten ID (die größer als 0 ist) ausgibt, wodurch die fortlaufende Inkrementierung durch den `TRIGGER` bestätigt wird.
Den `TRIGGER prevent_future_grosses` haben wir durch einen `INSERT`-Befehl getestet, welcher einem Film, dessen Veröffentlichungsdatum noch nicht angegeben wurde, positive Kinoverkaufszahlen zuordnet, woraufhin wie erwartet die Exception geworfen wird.

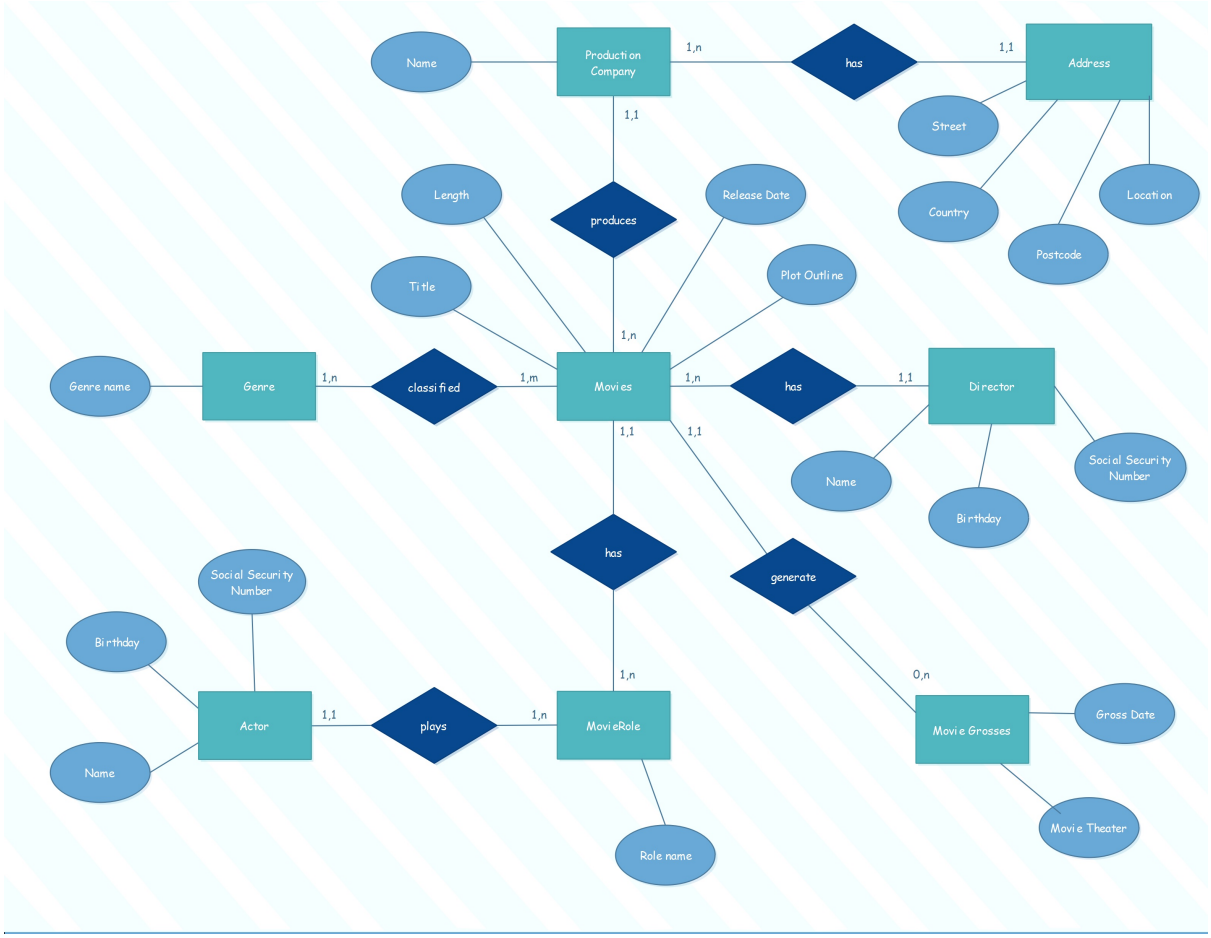
Nutzung von Indizes

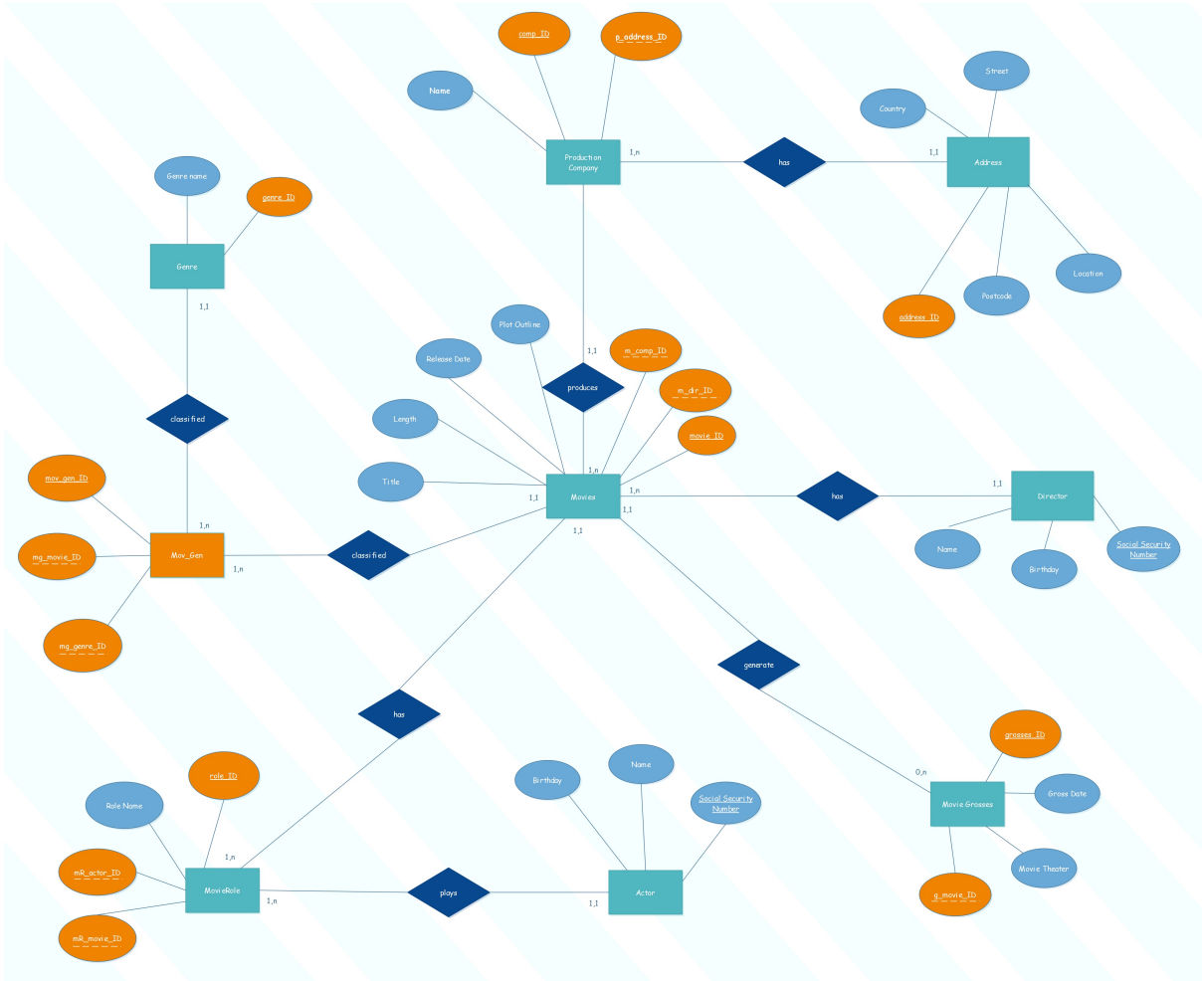
Wir haben uns gegen eine Index entschieden, da wir in jeder Tabelle durch die Verwendung eines Primärschlüssels eine implizite Indexierung der Daten vornehmen.
Ein expliziter Index würde sich jedoch bspw. in der Tabelle `movie_grosses` eignen, da diese durch fortlaufendes Eintragen von Umsätzen sehr schnell wachsen würde und hier außerdem wiederholte bestimmte Lesevorgänge vorkommen würden.

Quellen

Lediglich für den Increment-`TRIGGER` haben wir Hilfe aus dem Internet genutzt (https://stackoverflow.com/questions/11296361/how-to-create-id-with-auto-increment-on-oracle). Ansonsten haben wir für das gesamte SQL-Skript ausschließlich die in der VL und den Übungen gewonnenen Kenntnisse verwendet.

Logisches & Physisches ER-Diagramm





(Primärschlüssel = unterstrichen, Fremdschlüssel = gestrichelt)

Bewertungsschema (bitte als letzte Seite der Ausarbeitung einbinden):

Kriterium		**max.**	**ist**	**Kommentar**
Relative Qualität		15		
Spezifikation / Systembeschreibung		15		
ER-Modell inkl. formaler Korrektheit im Sinn der Normalisierung		10		
Sinnvolle Nutzung von Normierungsdaten		5		
Lauffähigkeit		5		
Restartfähigkeit		5		
Vollständigkeit der Testdaten		5		
Umfang der Umsetzung des Datenmodells		-	-	
	Anzahl der Tabellen	5		
	Vollständige Nutzung der Datentypen	5		
	Sinnvolle Nutzung von Randbedingungen	5		
	Sinnvolle Nutzung von Indexierung	5		
Summe				