Praktikum Datenbanken / DB2
Woche 7: Trigger, benutzerdefinierte Funktionen und rekursive Anfragen

Betreuer: Gudrun Fischer, Tobias Tuttas, Camille Pieume
Raum: LF 230
Bearbeitung: 26., 27. und 29. Juni 2006

Datum	
Team (Account)	
Vorbereitung	
Präsenz	

Aktuelle Informationen, Ansprechpartner und Material unter:
`http://www.is.informatik.uni-duisburg.de/courses/dbp_ss06/index.html`

Wochenziele

In dieser Woche geht es um `Trigger`, benutzerdefinierte Funktionen und rekursive Anfragen. Für jedes dieser Konzepte wollen wir Beispiele entwickeln.

Trigger

Als Trigger (deutsch Auslöser) bezeichnet man in SQL eine Anweisungsfolge (eine Abfolge von Aktionen), die ausgeführt wird, wenn eine verändernde Anweisung auf einer bestimmten Tabelle ausgeführt werden soll. Wir erinnern uns aus früheren Übungen, dass verändernde Anweisungen für Tabellen DELETE, INSERT und UPDATE sind.

Man kann Trigger zum Beispiel nutzen, um

- neu eingefügte oder zu verändernde Tupel auf ihre Gültigkeit bezüglich vorgegebener Bedingungen zu überprüfen,
- Werte zu generieren,
- in anderen Tabellen zu lesen (etwa zur Auflösung von Querverweisen) oder zu schreiben (etwa zur Protokollierung)

Insbesondere kann man mit Triggern Regeln in der Datenbank modellieren. Einen Trigger erstellt man mit dem CREATE TRIGGER-Statement, mit DROP TRIGGER wird er gelöscht.

db2s2e81.pdf, S. 414

Man kann Trigger entweder so schreiben, dass sie für jedes zu ändernde Tupel einmal (FOR EACH ROW) oder für jedes verändernde Statement einmal (FOR EACH STATEMENT) ausgeführt werden. Letzteres ist allerdings nur für die sogenannten AFTER-Trigger erlaubt, die im Anschluß an eine verändernde Operation tätig werden. In diesem Fall wird der Trigger auch dann abgearbeitet, wenn kein Tupel von dem DELETE- oder dem UPDATE-Statement betroffen ist.

Über die REFERENCING-Klausel werden Namen für die Übergangsvariablen festgelegt: `OLD AS name` bzw. `NEW AS name` definiert `name` als Name für die Werte des betroffenen Tupels vor bzw. nach der Ausführung der auslösenden Operation. Entsprechend definieren `OLD_TABLE AS identifier` bzw. `NEW_TABLE AS identifier` diesen als Name für eine hypothetische Tabelle, welche die betroffenen Tupel vor bzw. nach der auslösenden Operation enthält. Dabei gilt :

auslösendes Ereignis und Zeitpunkt	**ROW-Trigger darf benutzen**	**STATEMENT-Trigger darf benutzen**
BEFORE INSERT	NEW	-
BEFORE UPDATE	OLD, NEW	-
BEFORE DELETE	OLD	-
AFTER INSERT	NEW, NEW_TABLE	NEW_TABLE
AFTER UPDATE	OLD, NEW, OLD_TABLE, NEW_TABLE	OLD_TABLE, NEW_TABLE
AFTER DELETE	OLD, OLD_TABLE	OLD_TABLE

Über vorhandene Trigger auf einer Datenbank kann man die System-Tabelle `SYSIBM.SYSTRIGGERS` konsultieren:

```
SELECT name,text
FROM SYSIBM.SYSTRIGGERS
```

Beispiel:

Angenommen, wir führten eine Tabelle `afps` (Anzahl Folgen pro Serie), dann könnte ein Trigger für das Einfügen von neuen Tupeln in die Tabelle `serienfolge` eventuell derart aussehen:

```
CREATE TRIGGER neue_folge
   AFTER INSERT ON serienfolge
   REFERENCING NEW AS neu
   FOR EACH ROW MODE DB2SQL
   UPDATE afps
      SET anzahl_folgen = anzahl_folgen + 1
      WHERE neu.serientitel = afps.serientitel
        AND neu.drehjahr = afps.drehjahr
```

Und ein Einfügen eines Tupels in `serienfolge` würde automatisch zu einem UPDATE der Tabelle `afps` führen:

```
INSERT INTO afps (serientitel, drehjahr, anzahl_folgen)
VALUES ('Babylon 5 - The Telepath Tribes', 0);

INSERT INTO serienfolge (serientitel, drehjahr, staffelnr,
        folgennr, folgentitel)
VALUES ('Babylon 5 - The Telepath Tribes', 2007, 1, 1,
        'Forget Byron');

INSERT INTO serienfolge (serientitel, drehjahr, staffelnr,
        folgennr, folgentitel)
VALUES ('Babylon 5 - The Telepath Tribes', 2007, 1, 2,
        'Bester''s Best');

SELECT anzahl_folgen AS Anzahl
FROM afps
WHERE land='Babylon 5 - The Telepath Tribes'

ANZAHL
------
     2
```

Noch ein Beispiel:

```
CREATE TRIGGER check_ausleihe
   NO CASCADE BEFORE INSERT ON ausleihe
   REFERENCING NEW AS neu
   FOR EACH ROW MODE DB2SQL
   IF neu.ausleihdatum > current date
   THEN
      SIGNAL SQLSTATE '75000'
      SET MESSAGE_TEXT =
        'Ein Ausleihvorgang kann nicht im voraus eingefügt werden.';
   END IF
```

Dieser Trigger verbietet das Einfügen neuer Tupel in `ausleihe`, bei denen das

Ausleihdatum in der Zukunft liegt. (Es sollen also nur bereits stattgefundene Ausleihvorgänge eingetragen werden.) Der Versuch führt zu einem Fehler und das Statement wird nicht ausgeführt.

Benutzerdefinierte Funktionen

Über User Defined Functions (UDFs) lassen sich neue Funktionen (siehe Woche 5) implementieren. So läßt sich das DBMS um anwendungsfallspezifische Funktionalitäten erweitern. Es wird dabei zwischen externen UDFs und SQL-UDFs unterschieden. Im Rahmen des Praktikums sollen nur letztere behandelt werden. SQL-UDFs werden in der SqlPl (*SQL Programming Language*) implementiert.

Man unterscheidet dabei zwischen UDFs mit skalarer, tupelwertiger und tabellenwertiger Rückgabe. Skalare UDFs liefern einen skalaren Wert zurück und lassen sich in Ausdrücken verwenden. Tupelwertige UDFs sind für benutzerdefinierte Typen interessant, und bleiben hier zunächst außen vor. Tabellenwertige UDFs können wie eine normale Tabelle in einer FROM-Klausel benutzt werden.

SQL-UDFs können auf die Datenbank zugreifen und dürfen selbst SQL-Statements enthalten, dabei besitzen temporale Registerwerte innerhalb einer SQL-UDF stets einen konstanten Wert (d.h. tritt ein Zeitregister mehrfach in einer SQL-UDF auf, so hat es bei jedem Vorkommen den selben Wert).

Mit Hilfe der Systemtabelle `SYSCAT.FUNCTIONS` kann man sich einen Überblick über vordefinierte und existierende benutzerdefinierte Funktionen verschaffen:

```
-- alle Funktionen
SELECT funcschema, funcname
FROM syscat.functions

-- nur benutzerdefinierte Funktionen
SELECT funcschema, funcname
FROM syscat.functions
WHERE definer <> 'SYSIBM'
```

Die Tabelle `SYSCAT.FUNCTIONS` enthält neben Namen und Schema der Funktionen noch eine Reihe weiterer Informationen, wie man leicht sieht, wenn man `describe table SYSCAT.FUNCTIONS` ausführt.

Erstellt wird eine benutzerdefinierte Funktion über das CREATE FUNCTION-Statement. Bevor wir uns einige Beispiele ansehen, zunächst eine kurze Einführung in die grundlegenden Konstrukte von SqlPl.

db2s2e81.pdf, S. 188, S. 254ff

SQL Programming Language

Die SqlPl ist im SQL99-Standard definiert, und ist eine sehr einfache Programmiersprache, die innerhalb eines DBMS benutzt werden kann. Insbesondere findet sie in Stored Procedures, Triggers (siehe weiter oben) und bei der Implementierung von SQL-UDFs Verwendung. Im Falle von Triggern und Funktionen

werden die SqlPl-Anweisungen im DBMS abgelegt und bei Bedarf interpretiert.

Dabei umfasst die SqlPl unter anderem die im Folgenden kurz beschriebenen Anweisungen. Prozeduren erlauben zusätzliche Kontroll-Statements, die hier aber zunächst nicht behandelt werden sollen. **Getrennt werden die einzelnen Anweisungen in SqlPl durch Semikola. Daher muß man beachten, dass man Zeilen im CLP nicht mit einem Semikolon enden läßt (mit Ausnahme des tatsächlichen Ende des Statements), oder ein alternatives Terminierungszeichen benutzen, z.B. durch Aufruf des CLP als** `db2 -c -td@`.

Um mehrere SqlPl-Statements zusammenzufassen, kann man im Rumpf einer Trigger- oder Funktionsdefinition einen mit BEGIN ... END geklammerten Anweisungsblock benutzen. Dieses ist ausführlicher in der Dokumentation beschrieben, und sieht z.B. so aus:

db2s2e81.pdf, S. 123

```
name:
BEGIN ATOMIC
     DECLARE counter INTEGER;
     Anweisungen;
END name
```

Durch das Schlüsselwort ATOMIC wird festgelegt, dass bei Fehlern innerhalb des Anweisungsblocka **alle** Anweisungen des Blocks zurückgenommen werden. Variablen, die man innerhalb des Anweisungsblocks benutzen möchte, definiert man mit DECLARE. Dabei gibt man der Variable einen lokalen Namen, spezifiziert den Datentyp und definiert gegebenenfalls einen Default-Wert für die Variable (standardmäßig NULL).

Durch das Label *name* ist es möglich, den benannten Block durch ein LEAVE-Statement zu verlassen. Außerdem kann das Label zur Qualifizierung von Variablen benutzt werden.

SQL-Statements

Viele der üblichen SQL-Statements sind auch innerhalb von Funktionsdefinitionen erlaubt: das Fullselect, UPDATE oder DELETE mit Suchbedingung, INSERT und SET für Variablen.

FOR

db2s2e81.pdf, S. 775

Das FOR-Statement führt ein oder mehrere SQL-Statements für jedes Tupel einer Relation aus. Dabei kann die FOR-Schleife durch Voranstellung von *label:* benannt werden. Dieses Label kann dann in LEAVE- oder ITERATE-Statements benutzt werden.

```
BEGIN
    DECLARE vollertitel CHAR(40);
    FOR t AS
       SELECT titel, jahr, budget FROM produktion
```

```
      DO
        SET vollertitel = t.titel || ' (' || t.jahr || ')';
        INSERT INTO einetabelle VALUES (vollertitel, t.budget);
    END FOR;
END
```

Das obige Beispiel benutzt den '||'–Operator zur Stringkonkatenation (Zusammenfügen zweier Zeichenketten zu einer neuen).

IF

db2s2e81.pdf, S. 782

Das IF-Statement ermöglicht bedingte Ausführung von Statements. Die Anweisung wird ausgeführt, wenn die Suchbedingung zu *true* evaluiert. Liefert sie *false* oder *unknown*, dann wird stattdessen die nächste Suchbedingung hinter einem ELSEIF ausgeführt. Evaluiert keine Suchbedingung zu *true*, wird stattdessen der ELSE-Zweig ausgeführt.

```
IF anzahl_postings < 10 THEN
   SET typ = 'Anfänger';
ELSEIF azahl_postings < 100 THEN
   SET typ = 'Mitglied';
ELSEIF anzahl_postings < 1000 THEN
   SET typ = 'Seniormitglied';
ELSE
   SET typ = 'Profi';
END IF

IF folgen_gesamt > (SELECT count(*)
        FROM seriefolge
        WHERE serientitel = titel1
          AND drehjahr = titel2)
THEN RETURN 'unvollständig';
ELSE RETURN 'vollständig';
END IF
```

WHILE

db2s2e81.pdf, S. 799

Eine WHILE-Schleife wird solange durchlaufen, bis die angegebene Suchbedingung nicht mehr als *true* evaluiert. Bei jedem Durchlauf wird der Schleifenkörper ausgeführt. Dabei kann die WHILE-Schleife durch Voranstellung von *label:* benannt werden. Dieses Label kann in LEAVE- oder ITERATE-Statements benutzt werden.

```
WHILE counter < 10
DO
   Schleifenkörper
END WHILE
```

ITERATE

db2s2e81.pdf, S. 784

Mit dem ITERATE-Statement verläßt man die Abarbeitung des aktuellen Schleifendurchlaufs und springt zum Anfang der benannten Schleife (FOR, LOOP, REPEAT oder WHILE) zurück.

```
ITERATE schleifenname
```

LEAVE

db2s2e81.pdf, S. 785

Mit dem LEAVE-Statement verläßt man vorzeitig eine benannte Schleife (FOR, LOOP, REPEAT, WHILE) oder einen Anweisungsblock. Dabei werden gegebenenfalls alle Cursor geschlossen, mit Ausnahme derer, welche die Ergebnisrelation definieren. (Über Cursor mehr in der nächsten Woche.)

```
LEAVE schleifenname
```

RETURN

db2s2e81.pdf, S. 794

Mit dem RETURN-Statement kann man aus einer Funktion heraus springen. Dabei muß ein Rückgabewert definiert werden, der mit dem Rückgabetyp der Funktion kompatibel ist. Bei Tabellenfunktionen muß ein Fullselect benutzt werden.

```
RETURN substring(name,locate(name,',',))

RETURN SELECT name, todesdatum
       FROM person
       WHERE todesdatum IS NOT NULL
```

Zuweisungen

Über SET kann man innerhalb eines Anweisungsblock oder im Rumpf eines Kontroll-Statements Zuweisungen an Variables tätigen.

SIGNAL

db2s2e81.pdf, S. 796

Mit dem SIGNAL-Statement kann man einen SQL-Fehler oder eine SQL-Warnung werfen. Der erklärende Fehlertext darf maximal 70 Zeichen lang und kann auch eine Variable sein. Dabei gilt zur Benennung des SQL-Status:

- ein SQL-Status ist ein fünf Zeichen langer String aus Großbuchstaben und Ziffern,
- er darf nicht mit '00' beginnen (Erfolg),
- in Triggern oder Funktionen darf er nicht mit '01' (Warnung) oder '02' beginnen,

- ein SQL-Status, der nicht mit '00', '01' oder '02' beginnt, signalisiert einen Fehler,
- beginnt der String mit 0-6 oder A-H, so müssen die letzten drei Zeichen mit I-Z beginnen

```
SIGNAL SQLSTATE  '03ILJ'
   SET MESSAGE_TEXT='Fehler: Illegales Jahr'
```

Beispiele

Skalare Funktion: Tangens

Ein einfaches Beispiel für eine skalare Funktion ist die Berechnung des Tangens (aus der IBM DB2 Dokumentation):

```
CREATE FUNCTION tan (x DOUBLE)
   RETURNS DOUBLE
   LANGUAGE SQL
   CONTAINS SQL
   DETERMINISTIC
   RETURN sin(x)/cos(x)
```

Diese Funktion berechnet den Tangens zu einem Wert. Sie erwartet einen Eingabewert vom Datentyp DOUBLE, der intern den Variablennamen `x` erhält und liefert einen Ausgabewert ebenfalls vom Datentyp DOUBLE, der als Quotient aus den existierenden Funktionen Sinus und Kosinus berechnet wird.

RETURNS definiert die erwartete Rückgabe, hier ein skalarer Wert vom Typ DOUBLE

LANGUAGE SQL signalisiert, dass die Funktion in SQL geschrieben ist

CONTAINS SQL signalisiert, dass die Funktion nur nicht-lesende und nicht-modifizierende SQL-Statements benutzen darf, READS SQL DATA erlaubt auch lesende SQL-Statements

DETERMINISTIC ist eine optionale Klausel und signalisiert, dass bei Aufruf mit den gleichen Parametern stets das gleiche Resultat geliefert wird (sonst benutzt man NON DETERMINISTIC)

RETURN liefert schließlich das Ergebnis einer Berechnung zurück

Aufgerufen würde die Funktion z.B. derart:

```
SELECT tan(c)
FROM (values (1)) t(c)
```

Tabellenwertige Funktion

Das letzte Beispiel zeigt eine tabellenwertige Funktion, die für einen bestimmten Film alle Schauspieler mit deren Namen und Vornamen liefert.

```
CREATE FUNCTION schauspielerliste (filmtitel VARCHAR(200),
    filmjahr INTEGER)
  RETURNS TABLE (name VARCHAR(100),vorname VARCHAR(100))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
     SELECT name,vorname
     FROM schauspieler, spieltinfilm
     WHERE schauspieler.id = spieltinfilm.schauspielerid
       AND spieltinfilm.titel = filmtitel
       AND spieltinfilm.drehjahr = filmjahr
```

RETURNS legt das Aussehen und Format der Rückgabetabelle fest

READS SQL DATA ist nötig, da wir in dieser Funktion lesend auf zwei Tabellen, nämlich `schauspieler` und `spieltinfilm` zugreifen

RETURN liefert nun das Ergebnis eines SELECTs zurück

Eine tabellenwertige Funktion kann normal im FROM-Teil eines SELECT-Statements benutzt werden, wenn man sie in eine `TABLE`–Klausel einschließt. Die Syntax sieht im Beispiel wie folgt aus:

```
SELECT *
FROM TABLE (schauspielerliste('Event Horizon',1997)) sl;
```

Rekursive Anfragen

In einem vollen SELECT-Statement ist es auch möglich, rekursive Anfragen zu stellen. Rekursive Anfragen sind solche Anfragen, die sich in ihrer Definition auf sich selbst beziehen. Derart kann man etwa die *transitive Hülle* eines Tupels oder Stücklisten berechnen.

db2s1e81.pdf, S. 601

Ein kurzer Ausflug, um das Konzept der Rekursion zu erklären:

> **Rekursion**
> *aus Wikipedia, der freien Enzyklopädie*
> (`http://de.wikipedia.org/wiki/Rekursion`)
>
> Rekursion bedeutet Selbstbezüglichkeit. Sie tritt immer dann auf, wenn etwas auf sich selbst verweist oder mehrere Dinge aufeinander, so dass merkwürdige Schleifen entstehen. So ist z.B. der Satz "Dieser Satz ist unwahr" rekursiv, da er von sich selber spricht. [...]
>
> Dabei ist Rekursion ein allgemeines Prinzip zur Lösung von Problemen, das in vielen Fällen zu "eleganten" mathematischen Lösungen führt. Als Rekursion bezeichnet man den Aufruf bzw. die Definition einer Funktion durch sich selbst. Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen sog. infiniten Regress [Endlosrekursion].

> [...] Die Definition von rekursiv festgelegten Funktionen ist eine grundsätzliche Vorgehensweise in der funktionalen Programmierung. Ausgehend von einigen gegebenen Funktionen (wie zum Beispiel unten die Summen-Funktion) werden neue Funktionen definiert, mithilfe derer weitere Funktionen definiert werden können.
>
> Hier ein Beispiel für eine Funktion $sum : N_0 \rightarrow N_0$, die die Summe der ersten n Zahlen berechnet:
>
> $$sum(n) = \begin{cases} 0, & \text{falls } n = 0 \text{ (Rekursionsbasis)} \\ sum(n-1) + n, & \text{falls } n \neq 1 \text{ (Rekursionsschritt)0} \end{cases}$$

Bei einer rekursiven Anfrage (RA) muß man folgende Regeln beachten:

- Jedes Fullselect, das Teil einer RA ist, muß mit SELECT beginnen (aber nicht mit SELECT DISTINCT). Es darf nicht geschachelt sein. Als Vereinigung **muss** UNION ALL benutzt werden.
- Im WITH-Teil müssen Spaltennamen explizit angegeben werden.
- Das erste Fullselect in der ersten Vereinigung darf sich nicht auf die zu definierende Tabelle beziehen. Es bildet die Rekursionsbasis.
- Die Datentypen und Längen der Spalten der zu definierenden Tabelle werden durch die SELECT-Klausel der Rekursionsbasis festgelegt.
- Kein Fullselect in der rekursiven Definition darf eine Aggregatfunktion, eine GROUP BY- oder eine HAVING-Klausel enthalten. Die FROM-Klauseln in der rekursiven Definition dürfen höchstens eine Referenz auf die zu definierende Tabelle besitzen.
- Es dürfen keine Subqueries verwendet werden.

Wenn die Daten Zyklen enthalten können, dann ist es möglich, dass durch eine RA eine Endlosrekursion erzeugt wird. Dann ist es wichtig, eine Abbruchbedingung zu definieren:

- Die zu definierende Tabelle muß ein Attribut enthalten, das durch eine INTEGER-Konstante initialisiert wird, und regelmäßig erhöht wird.
- In jeder WHERE-Klausel muß eine Prüfbedingung auftreten.

Die Auswertung einer rekursiven Anfrage läuft wie folgt ab: Zuerst werden die Tupel der Rekursionsbasis berechnet, dann werden ausgehend von diesen Tupeln gemäß des zweiten Teils des Fullselects (nach dem UNION ALL) weitere Tupel berechnet. Dabei werden jedoch nur diejenigen Tupel aus der rekursiv definierten Relation verwendet, die beim letzten Berechnungsschritt neu hinzugekommen sind (beginnend also mit den Tupeln der Rekursionsbasis). Kamen in einem Berechnungsschritt keine neuen Tupel hinzu, so endet die Berechnung.

Beispiel:

Gegeben sei eine Tabelle mit Bauteilen, in denen die Komponten für Werkstücke festgehalten werden. Dabei können diese Komponenten selber wiederum aus Einzelteilen zusammengesetzt sein.

```
CREATE TABLE bauteile (
       stueck     VARCHAR(8),
       komponente VARCHAR(8),
       menge      INTEGER);
```

Um nun zu ermitteln, welche Basiskomponenten insgesamt nötig sind, um ein Beispiel-Bauteil herzustellen, kann man eine rekursive Anfrage benutzen. In der folgenden RA ist die im ersten Teil der WITH-Klausel definierte Query `rek` die Rekursionsbasis, im zweiten Teil der WITH-Klausel folgt dann die Definition der Rekursion. Die so rekursiv definierte Ergebnisrelation wird dann im SELECT DISTINCT-Teil benutzt.

```
WITH rek (stueck, komponente, menge) AS (
        SELECT r.stueck, r.kompenente, r.menge
        FROM bauteile r
        WHERE r.stueck = 'Beispiel-Bauteil'
   UNION ALL
        SELECT kind.stueck, kind.komponente, kind.menge
        FROM rek vater,
             bauteile kind
        WHERE vater.komponente = kind.stueck
     )
SELECT DISTINCT stueck, komponente, menge
FROM rek
ORDER BY stueck, komponente, menge
```

Übersicht über die neuen Befehle

create trigger *name* ...	erstelle einen neuen Trigger *name*
drop trigger *name* ...	lösche den Trigger *name*
create function *name* ...	erstelle eine neue Funktion *name*
drop function *name*	lösche die Funktion *name*

Aufgaben

Vorbereitung (Hausaufgaben)

V1: Präsenzaufgaben vorbereiten

Bereitet die Präsenzaufgaben P1 bis P4 sorgfältig vor, damit Ihr sie in der Präsenzzeit vollständig umsetzen könnt. Überlegt Euch zu jeder Aufgabe,

- wie Ihr sie lösen wollt,
- welche SQL- oder SqlPl-Konstrukte Ihr dafür braucht,
- und welche Syntax diese Konstrukte haben.

Schreibt Euch den Code für Eure Lösungen schon vor der Stunde auf und bringt diese Aufzeichnungen mit ins Praktikum.

Präsenz

P1: Trigger zum Überprüfen von Telefonnummern

Schreibt einen Trigger `checkFon`, um beim Einfügen von Telefonnummern deren Syntax zu überprüfen. Zulässige Telefonnummern sollen ausschließlich aus den Zeichen '/', '␣' (Leerzeichen), '-' und Ziffern bestehen.

P2: Skalare Funktion

Schreibt eine *skalare* Funktion `verfuegbar`, die `true` oder `false` zurückgibt, je nachdem ob ein Medium gerade verfügbar ist oder nicht. Jedes Medium, das gerade nicht verliehen ist, soll dabei als verfügbar gelten.

P3: Tabellenwertige Funktion

Schreibt eine `tabellenwertige` Funktion `ausgeliehenVonNick`, die zu einem gegebenen Nick alle Medien auflistet, die der betreffende Anwender ausgeliehen und noch nicht zurückgegeben hat.

P4: Rekursive Anfrage

In der Beispieldatenbank `FILME` wurden zwei neue Tabellen angelegt:

```
CREATE TABLE Kategorie (
  Id INTEGER NOT NULL
    GENERATED BY DEFAULT AS IDENTIY
    PRIMARY KEY,
  Name VARCHAR(40) NOT NULL,
  Supercat INTEGER
```

```
    REFERENCES Kategorie(Id)
      ON DELETE SET NULL
)

CREATE TABLE FilmKategorie (
  Titel VARCHAR(200) NOT NULL,
  Drehjahr INTEGER NOT NULL,
  Kategorie INTEGER NOT NULL
    REFERENCES Kategorie(Id)
      ON DELETE CASCADE,
  PRIMARY KEY (Titel,DRehjahr,Kategorie)
)
```

Es gibt 14 Kategorien, die teilweise noch weitere Unterkategorien besitzen. Jede Kategorie kann jedoch nur zu höchstens einer Oberkategorie gehören.

Einige Filme wurden kategorisiert. Dabei kann jeder Film mehreren Kategorien zugeordnet sein. Es ist immer nur die spezifischste Kategorie angegeben, ein Film in einer Unterkategorie gehört jedoch automatisch auch zu deren Oberkategorie.

Beispielsweise ist der Film „My Girl" aus dem Jahre 1991 in der Kategorie „Erste Liebe" (Nummer 1401) eingeordnet. Das dazugehöreige Tupel lautet:
`('My Girl',1991,1401)`

Die Kategorie „Erste Liebe" ist jedoch unter der Kategorie „Kinder" (Nummer 14) angesiedelt. Damit gehört der Film „My Girl" automatisch auch zu den Kinderfilmen, obwohl es kein explizites Tupel `('My Girl',1991,14)` gibt.

Aufgabe:

Schreibt eine Anfrage, die zu dem Film „Galaxy Quest" von 1999 *alle* Kategorien zurückliefert (inklusive der Oberkategorien).

Freiwillige Aufgaben zur Prüfungsvorbereitung

Die folgenden Aufgaben dienen der Prüfungsvorbereitung und zum Weiterdenken. Sie sind wie die Pflichtaufgaben klausur- bzw. prüfungsrelevant.

Diese Aufgaben müssen nicht unbedingt in dieser Woche gelöst werden, wir empfehlen jedoch, sie auf jeden Fall zu bearbeiten!

F1: Verwendung von Triggern

(a) Überlegt Euch mindestens zwei weitere Beispiele in Eurer eigenen Datenbank, wo Ihr Trigger sinnvoll einsetzen könntet.

(b) Vergleicht Trigger mit Constraints. Werden beide für dieselbe Zwecke verwendet, oder gibt es Unterschiede? Wann ist es sinnvoller, mit Triggern statt mit Constraints zu arbeiten?

F2: Verwendung von Funktionen

Welche Unterschiede seht Ihr in der Verwendung von Triggern bzw. benutzerdefinierten Funktionen? Wann sind Trigger sinnvoller, wann eigene Funktionen? Gibt es in Eurer eigenen Datenbank weitere sinnvolle Anwendungensmöglichkeiten für skalare oder für tabellenwertige Funktionen?

F3: `WITH ... SELECT`-Statement

Macht Euch mit Hilfe der angegebenen Handbuchabschnitte, bzw. der Online-Dokumentation oder anderer Literatur mit der Syntax des WITH...SELECT-Statements vertraut.

Was sind andere Nutzen für einen in der `WITH`-Klausel definierten Tabellenausdruck außer der Verwendung in rekursiven Anfragen?

F4: Rekursion und transitive Hülle

Betrachtet die Tabellen aus Aufgabe P4.

(a) Definiert eine Sicht (View), die zu einem beliebigen Film *alle* Kategorien und deren Nummern liefert.

(b) Was ist bei der vorhergehenden Teilaufgabe die *transitive Hülle*? Worauf bezieht sich der Ausdruck *transitive Hülle*?

Zum Schluss ...
stoppt bitte Eure Datenbank-Manager-Instanz auf `salz`,
loggt Euch von `salz` und vom lokalen Rechner aus,
aber schaltet den Rechner nicht ab.