

Praktikum Datenbanken / DB2

Woche 8: User Defined Functions, SQL Programming Language

Raum: LF 230

Nächste Sitzung: 1./4. Dezember 2003

Die Dokumentation zu DB2 steht online zur Verfügung. Eine lokale Installation der Dokumentation findet sich unter der Adresse http://salz.is.informatik.uni-duisburg.de/db2doc/de_DE/index.htm.

Die englischsprachigen Handbücher für IBM DB2 V8.1 liegen als PDF Dateien lokal im Verzeichnis `/usr/projects/db2doc/`. Diese Handbücher sind sehr umfangreich und sollten nur zum Betrachten am Bildschirm herangezogen und nicht ausgedruckt werden. Die Referenzen in diesem Arbeitsblatt beziehen sich auf diese Handbücher. Die gleichen Informationen sind jedoch auch in der Online-Dokumentation zu finden.

User Defined Functions

Über User Defined Functions (UDFs) lassen sich neue Funktionen (siehe Woche 5) implementieren. So läßt sich das DBMS um anwendungsfallsspezifische 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 (der 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).

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 (siehe nächste Woche), Triggers (letzte Woche) 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 Anweisungsblock **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
```

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 budget < 1000 THEN
    SET typ = 'Low Budget';
```

```
ELSEIF budget < 100000 THEN
    SET typ = 'C-Movie';
ELSEIF budget < 1000000 THEN
    SET typ = 'B-Movie';
ELSE
    SET typ = 'Big Screen Movie';
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 herausspringen. Dabei muß ein Rückgabewert definiert werden, der mit dem Rückgabebetyp 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 03ILJ
SET MESSAGE_TEXT ‘‘Fehler: Illegales Jahr’’
```

Zwei Beispiele

Zunächst ein einfaches Beispiel für eine skalare Funktion 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)
```

Das zweite Beispiel zeigt eine tabellenwertige Funktion, die für eine gegebene Filmart (z.B. 'TV' oder 'VG') Titel und Jahr aus der Tabelle `film` zurückliefert.

```
CREATE FUNCTION filmtyp (art CHAR(2))
  RETURNS TABLE (titel VARCHAR(75), jahr INTEGER)
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
  SELECT titel, jahr
  FROM film
  WHERE film.art = filmtyp.art
```

RETURNS legt das Aussehen und Format der Rückgabetable fest

READS SQL DATA ist nötig, da wir in dieser Funktion lesend auf eine Tabelle, nämlich `film` zugreifen

RETURN liefert nun das Ergebnis eines SELECTs zurück

Eine tabellenwertige Funktion kann normal im FROM-Teil eines SELECT-Statements benutzt werden. Die Syntax sieht im Beispiel wie folgt aus:

```
SELECT *
FROM TABLE (filmtyp(char('VG'))) vg;
```

Vorüberlegungen

- Macht Euch anhand der angegebenen Abschnitte der DB2-Dokumentation mit der Syntax des CREATE FUNCTION-Statements und der SQL-Control Statements vertraut. Bearbeitet zur Vorbereitung die Aufgabe (f).
- Seht Euch zur Auffrischung noch einmal die Abschnitte zu vordefinierten Funktionen auf Blatt 5 und im Handbuch an.

Aufgaben

- Namen sind in der Filmdatenbank in der Form 'Nachname, Vorname(n)' gespeichert. Nun möchte das Management unseres DVD-Verleihs aber, dass Kunden in der Datenbank auch nach Vornamen und Nachnamen suchen können. Um dies zu erleichtern, schreibt zwei UDFs `vorname()` und `nachname()`, die zu Zeichenketten (Strings) des beschriebenen Formats, die Vor- bzw. Nachnamen zurückliefern. Ist der Eingabeparameter nicht in kompatibelem Format, soll `vorname()` den leeren String und `nachname()` die komplette Zeichenkette zurückliefern.
- Budgets sind in der Filmdatenbank als Zeichenkette gespeichert, die mit dem dreistelligen Währungscode beginnt und bei der Dreierblöcke von Ziffern durch Kommata getrennt werden, z.B. 'DEM 1,000,000'. Um die Budgets vergleichbar zu machen, schreibt eine UDF, welche einen solchen String in Euro umrechnet und als Integer zurückgibt.

Nehmt dazu folgende Umrechnungswerte an:

	1 €	
ATS	13.7603	Österreichischer Schilling
BEF	40.3399	Belgische Franc
CHF	1.5530	Schweizer Franken
CZK	31.7537	Tschechische Kronen
DEM	1.95583	Deutsche Mark
DKK	7.4377	Dänische Kronen
ESP	166.386	Spanische Peseta
FIM	5.94573	Finnmark
FRF	6.55957	Französischer Franc
GBP	0.6938	Britische Pfund
GRD	340.750	Griechische Drachmen
IEP	0.78756	Irishes Pfund
ITL	1936.27	Italienische Lira
JPY	129.2850	Japanische Yen
LUF	40.3399	Luxemburgische Franc
NLG	2.20371	Niederländischer Gulden
NOK	8,1638	Norwegische Kronen
PTE	200.482	Portugiesischer Escudo
SEK	8,9605	Schwedische Kronen
USD	1.1755	US Dollar

Es soll reichen, die Umrechnung für drei Währungen zu machen. In anderen Fällen soll die Funktion eine Fehlermeldung geben.

- (c) Benutzt die Währungsinformation des Attributs `budget` für eine UDF `herkunftsland()`.
- (d) Schreibt eine tabellenwertige SQL-UDF `kategorie()`, die zu einem Genrenamen (als String) alle Tupel aus der Tabelle `film` (!) zurückliefert, die zu diesem Genre gehören.
- (e) Testet Eure Funktionen!
- (f) Nun da wir mehr über SQLPL-Konstrukte wissen, überarbeitet die in der letzten Woche definierten Zähltrigger (Beispiel 1, sowie Aufgaben *c* und *d*). Benutzt gegebenenfalls einen Anweisungsblock, um die Update-Trigger zu einem einzelnen Trigger zusammenzufassen. Behandelt durch Fallunterscheidung den Fall, dass in `fps` noch kein Tupel für einen Schauspieler existiert.