Praktikum: Datenbanken
Woche 8: Benutzerdefinierte Funktionen

Raum: LF230 Abgabe bis **13.-15.6. 2005**

Datum	
Gruppe	
Vorbereitung	
Präsenz	

User Defined Functions

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 (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).

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 (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
```

Das 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 bevoelkerung < 1000 THEN
   SET typ = 'Dorf';
ELSEIF budget < 100000 THEN
   SET typ = 'Kleinstadt';
ELSEIF budget < 1000000 THEN
   SET typ = 'Großstadt';
ELSE
   SET typ = 'Metropole';
END IF

IF 0 < (SELECT count(*)
        FROM sym_grenze
        WHERE land1=erstesArgument
          AND land2=zweitesArgument)
THEN RETURN 'Nachbarländer';
ELSE RETURN 'keine Nachbarn';
```

```
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

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)
```

Skalare Funktionen: Boolesche Operatoren

DB2 kennt keinen Datentyp Boolean für Wahrheitswerte. Wir können uns aber eigene Funktionen schreiben, um diese Lücke zu schließen. Dazu kann man den Datentyp `SMALLINT` benutzen mit 1 für `TRUE` und 0 für `FALSE`.

Die booleschen Operatoren `AND`, `NOT` und `OR` könnten dann durch folgende skalare Funktionen ausgedrückt werden:

```
CREATE FUNCTION bool_and(x smallint, y smallint)
    RETURNS SMALLINT
BEGIN ATOMIC
    IF x IS NULL OR  y IS NULL THEN
         RETURN NULL;
    ELSEIF  x =1 AND  y = 1 THEN
      RETURN 1 ;
    ELSE RETURN 0;
    END IF;
END

CREATE FUNCTION bool_not(x smallint)
    RETURNS SMALLINT
BEGIN ATOMIC
    IF x IS NULL THEN
        RETURN  NULL;
    ELSEIF  x=1 THEN RETURN 0;
    ELSE RETURN 1;
   END IF;
END

CREATE FUNCTION bool_or(x smallint, y smallint)
    RETURNS SMALLINT
BEGIN ATOMIC
    IF x = 1  THEN  RETURN 1;
    ELSEIF x = 0 THEN
         RETURN  y;
    ELSEIF  y = 1  THEN RETURN 1;
```

```
    ELSE RETURN  NULL;
   END IF;
END
```

Die Schlüsselworte AND, OR und NOT dürfen nicht als Funktionsnamen benutzt werden, weshalb hier `bool_and`, `bool_or` und `bool_not` gewählt wurden. Man beachte, dass die Funktionen Werte vom Typ `SMALLINT` erwarten. Die Literale 1 und 0 werden standardmäßig als `INTEGER` interpretiert und müssen gegebenenfalls zunächst mittels der Funktion *smallint()* umgewandelt werden.

Das Skript zur Erstellung dieser Funktionen (`boolean_functions.sql`) liegt im Verzeichnis `/home/dbprak/scripts/`. Der Aufruf lautet `db2 -vtd@ -f boolean_functions.sql`.

Tabellenwertige Funktion

Das letzte Beispiel zeigt eine tabellenwertige Funktion, die für ein bestimmtes Land alle seine Städte mit Namen und Einwohnerzahl liefert.

```
CREATE FUNCTION staedteliste (land VARCHAR(100))
  RETURNS TABLE (name VARCHAR(35), einwohner INTEGER)
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
     SELECT stadt.name, stadt.bevoelkerung
     FROM stadt, land
     WHERE stadt.land = land.code
       AND land.name  = staedteliste.land
```

RETURNS legt das Aussehen und Format der Rückgabetabelle fest

READS SQL DATA ist nötig, da wir in dieser Funktion lesend auf zwei Tabellen, nämlich `stadt` und `land` 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 (staedteliste('Germany')) sl;
```

Übersicht über die neuen Befehle

create function *name* ...	erstelle eine neue Funktion *name*
drop function *name*	lösche die Funktion *name*

Aufgaben

Vorbereitungsaufgaben:

Aufgabe V8.1

Macht Euch anhand der angegebenen Abschnitte der DB2-Dokumentation mit der Syntax des CREATE FUNCTION-Statements und der SQL-Control Statements vertraut.

Aufgabe V8.2

Seht Euch zur Auffrischung noch einmal die Abschnitte zu vordefinierten Funktionen auf Blatt 5 und im Handbuch an.

Aufgabe V8.3

Schreibt eine Funktion, die zu einem Winkel als Double–Wert den Cotangens berechnet.

Aufgabe V8.4

Schreibt eine skalare Funktion, die einen Integer–Wert in eine binäre Zahl umwandelt. Also z.B. die Zahl 13 in '1101'. Das Ergebnis soll vom Typ `VARCHAR(32)` sein.

Aufgabe V8.5

Schreibt eine benutzerdefinierte Funktion `mementode`, die für die englischen Bezeichner für Arten von Mitgliedschaft in einer Organisation deutsche zurückliefert. In den Beispieldaten gibt es insgesamt 21 verschiedene Arten von Mitgliedschaft. Es genügt, die folgenden wichtigen Varianten zu übersetzen und für die übrigen die originale Zeichenkette zurückzuliefern:

```
applicant              Bewerber
associate member       assoziiertes Mitglied
guest                  Gast
member                 Vollmitglied
membership applicant   Bewerber
observer               Beobachter
```

Die Funktion soll benutzt werden, um eine Anfrage der folgenden Art zu schreiben:

```
SELECT land
FROM mitglied
WHHERE mementode(art) = 'Vollmitglied'
```

Präsenzaufgaben:

Aufgabe P8.6 Skalarwertige Funktionen

(a) Erstellt mit Hilfe des vorgegebenen Skriptes die booleschen Funktionen `bool_and`, `bool_not` und `bool_or` in Eurer Datenbank.

(b) Schreibt eine Funktion `ist_Metropole()`, die für einen Städtenamen *SMALLINT(0)* zurückliefert, wenn die Stadt existiert, aber weniger als 1 Million Einwohner hat. *SMALLINT(1)* soll zurückgeliefert werden, wenn die Stadt existiert und mindestens 1 Million Einwohner hat. *NULL* sonst.

(c) Schreibt eine Funktion `ist_Hauptstadt()`, die zu einem Städtenamen *SMALLINT(1)*, *SMALLINT(0)* oder *NULL* zurückliefert, wenn sie Hauptstadt eines Landes ist oder nicht.

(d) Schreibt eine Funktion `ist_HauptstadtMetropole()`, die zu einem Städtenamen *SMALLINT(1)*, *SMALLINT(0)* oder *NULL* zurückliefert, wenn sie Hauptstadt eines Landes und Metropole ist oder nicht. Benutzt dazu die zuvor definierten booleschen Funktionen.

Aufgabe P8.7 Tabellenwertige Funktion

Schreibt eine tabellenwertige Funktion `mitglieder()`, die zu einer Organisation (als String) und einen Kontinent (als String) alle Ländernamen (nicht Ländercodes) liefert, die Vollmitglied der Organisation sind und auf dem gegebenen Kontinent liegen. Benutzt werden soll die Funktion etwa wie folgt:

```
SELECT *
FROM TABLE (mitglieder('NATO','Europe')) as m
```

Aufgabe P8.8 Testen der Funktionen

Schreibt zu jeder Funktion geeignete Test-Statements und überprüft die Korrektheit Eurer Implementierung.