PL/pgSQL

VL Datenbanksysteme

Ingo Feinerer

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

Gliederung

PL/pgSQL-Programmteile in der Vorlesung

- Folien:
  - Enthalten viele Programmausschnitte
  - Programme manchmal nur auszugsweise auf den Folien wiedergegeben (immer nur die „wesentlichen“ Teile).
  - Durch das Auslassen von „unwesentlichen“ Details sind die Programme auf den Folien eventuell nicht lauffähig.
- PL/pgSQL-Sourcen:
  - Auf der DBS Homepage finden Sie die vollständigen Sourcen. Diese wurden unter PostgreSQL 8.4 getestet.

SQL vs. prozedurale Sprachen

- SQL
  - Deklarative Programmiersprache (*was*, nicht *wie*)
  - Vorteil: Bessere Optimierungsmöglichkeiten
  - Nachteil: „typische“ Sprachkonstrukte oft wünschenswert (SQL ist seit SQL:2003 Turing complete)
- Prozedurale DB-Programmiersprachen:
  - Fast alle DB-Systeme bieten eine (proprietäre) prozedurale Programmiersprache an
  - PostgreSQL: PL/pgSQL (inspiriert von Oracles PL/SQL)

Warum prozedurale DB-Sprachen?

- Performance: Netzwerktraffic wird eingespart
- Zwischenresultate, die der client nicht braucht, müssen nicht transferiert werden
- Mehrfache Parseläufe können unter Umständen vermieden werden
- Geschäftslogik findet sich an zentraler Stelle und kann von mehreren Applikationen genutzt werden
- Genaue Zugriffsbeschränkungen möglich

PostgreSQL

- PostgreSQL ermöglicht es *user-defined functions* in beliebigen Sprachen zu schreiben. Der Source der Funktion wird von PostgreSQL als Text behandelt und an den jeweiligen Adapter der Programmiersprache weitergegeben
- Standardmäßig unterstützt PostgreSQL auf diese Weise vier Sprachen: PL/pgSQL, PL/Tcl, PL/Perl und PL/Python. Zusätzlich lassen sich Sprachen wie PL/Java, PL/R oder PL/Ruby nachinstallieren.
- In dieser LVA werden wir nur PL/pgSQL behandeln. Diese Sprache ist Oracles PL/SQL äußerst ähnlich.

PostgreSQL

- User defined functions werden wie folgt erstellt:

Definition 1

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argname ] argtype [, ...] ] )
    [ RETURNS rettype
        | RETURNS TABLE ( colname coltype [, ...] ) ]
    AS $PROC$
        ... -- eigentlicher Source
    $PROC$ LANGUAGE plpgsql;
```

- Im Folgenden werden wir diesen „Rumpf“ oft auslassen.

Beispiel

Beispiel 2

```
CREATE OR REPLACE FUNCTION suche(matrnr numeric(10))
  RETURNS void AS $$
DECLARE
name     varchar(30);
semester numeric(2);
BEGIN
  SELECT s.name, s.semester INTO name, semester
    FROM students s WHERE s.matrnr = matrnr;
  IF (name IS NULL) THEN
    RAISE NOTICE 'Leider nichts gefunden';
  ELSE
    RAISE NOTICE 'Name: %, Semester: %', name, semester;
  END IF;
END;
$$ LANGUAGE plpgsql;

SELECT suche(0620611);
```

Aufbau eines PL/pgSQL Blocks

- DECLARE Section (optional): Variablen-, Typen- und Cursor-Deklarationen
- BEGIN Section: Enthält die eigentliche Programmlogik, z.B.: Zugriff auf die DB, Wertzuweisungen, Schleifen, Verzweigungen, etc.
- EXCEPTION Section (optional): Fehlerbehandlung. Tipp: Ein Block mit EXCEPTION Section bringt signifikante Performanceeinbußen mit sich.

Aufbau eines PL/pgSQL Blocks

- Kleinstmöglicher Block:

```
BEGIN
    -- mindestens ein Statement, z.b.
    NULL
END;
```

- Blöcke können beliebig geschachtelt werden.
  Interessante Anwendung: Innerer Block kann analog dem try . . . catch Konstrukt in Java genutzt werden

Quellen

- PostgreSQL Online Dokumentation:
  `http://www.postgresql.org/docs/current/`
- Folien dienen nur zum Überblick, Corner Cases und Details findet man in der Dokumentation.
- *UNBEDINGT* nach der Vorlesung korrespondierende Kapitel in der Dokumentation lesen!

Gliederung

Variablen-Deklaration

- Alle benutzten Variablen müssen in der DECLARE Sektion deklariert werden.
- Variablen können einen beliebigen SQL Typ haben (integer, varchar, char, . . .)
- Syntax:
  name [CONSTANT] type [**NOT NULL**] [(**DEFAULT** | :=) expression];

Beispiel 3

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Copying Types

Copying Types übernehmen den Datentyp einer anderen Variable oder Tabellenspalte

Beispiel 4

```
user_id  users.user_id%TYPE;
 seller_id   customer_id%TYPE;
```

- user_id hat denselben Typ wie die Spalte user_id in der users Tabelle.
- Vorteil: DRY[1] — Ändert sich der Datentyp der Spalte muss die Funktion nicht geändert werden.
- Hilft bei polymorphen Funktionen

[1] Don't repeat yourself

Row Types

- Eine sgn. *composite type* Variable speichert eine komplette Zeile eines SELECT oder FOR Queries.
- Die einzelnen Felder werden mittels Punkt-Notation angesprochen (rowvar.field)
- Nicht vorhandene Felder anzusprechen führt zu einem Runtime Error.

Beispiel 5

```
name table_name%ROWTYPE;
```

Record Types

- Ähnlich den Row Types, aber ohne vorgegebene Struktur.
- Schreibt man erstmalig ein Feld des Record Types wird dieses erstellt.

Beispiel 6

```
name RECORD;
```

Beispiel

Beispiel 7

```
DECLARE
    order order%ROWTYPE;
    shipped_item line_items%ROWTYPE;
    seller_id users.user_id%TYPE;
    customer_id seller_id%TYPE;
    shipping_information RECORD;
BEGIN
    -- mindestens ein Statement, z.b.
    NULL
END;
```

Variablensubstitution

- Jedes Token das einem Variablennamen entspricht wird ersetzt.
- D.h. das folgende Beispiel ist fehlerhaft

Beispiel 8

```
DECLARE
   val text;
BEGIN
   ...
   SELECT val FROM table INTO val
     WHERE key = 12345

   -- richtig:
   SELECT t.val FROM table t INTO val
     WHERE key = 12345
```

Gliederung

Wertzuweisung / Ausdrücke

- Zuweisungsoperator: $:=$
- Operatoren in Ausdrücken:
  - Arithmetische Operatoren: $+$, $-$, $*$, $/$
  - Vergleichsoperatoren: $=$, $>$, $<$, $>=$, $<=$
    ungleich: $!=$ oder $<>$
  - Logische Operatoren: AND, OR, NOT
  - Stringvergleiche: LIKE, NOT LIKE
    wildcards: %, _
  - String Konkatenation: $||$
  - Weitere SQL-Operationen:
    IS NULL, IS NOT NULL
    x BETWEEN a AND b, x IN (1,2,3)

Befehle ohne Result Set

- Alle SQL Befehle die kein result set zurückliefern können wie üblich aufgerufen werden.
- z.B. INSERT, UPDATE ohne RETURNING
- Aber *nicht* SELECT (ohne INTO)
- Variablen werden unterstützt

Beispiel 9

```
DECLARE
      key TEXT;
      delta INTEGER;
BEGIN
      ...
      UPDATE mytab SET val = val + delta
        WHERE id = key;
```

SELECT / PERFORM

- Manchmal möchte man eine Query ausführen, das Result Set aber verwerfen,
- z.B. wenn in der Query Funktionen mit Seiteneffekten aufgerufen werden.
- Innerhalb PL/pgSQL muss dafür PERFORM genutzt werden, ein SELECT ohne INTO führt zu einem Fehler.

Beispiel 10

```
PERFORM send_confirmation_email(id) FROM sales
WHERE status = 'new' AND sent_confirmation_email = FALSE;
```

Queries mit Single-Row Result

Das Ergebnis eines SQL Befehls mit einem einzeiligen Result Set kann einer RECORD oder ROWTYPE Variable zugewiesen werden, indem der SQL Befehl um die Klausel INTO erweitert wird:

Beispiel 11

```
SELECT expr INTO [STRICT] target FROM ...;
INSERT ... RETURNING expr INTO [STRICT] target;
UPDATE ... RETURNING expr INTO [STRICT] target;
DELETE ... RETURNING expr INTO [STRICT] target;
```

Wenn STRICT definiert ist muss die Query exakt eine Zeile zurückliefern.

Gliederung

Return Values einer Funktion

Definition 12

```
RETURN expression;
```

- Bei Funktionen mit Rückgabewert void und/oder out-Parametern kann RETURN ohne Argument genutzt werden.
- Alle anderen Funktionen *müssen* ein RETURN Statement haben. Eine Funktion die ohne RETURN endet führt zu einem Runtime Error.
- Siehe auch: RETURN NEXT, RETURN QUERY

IF-THEN

Definition 13

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

IF-THEN

Beispiel 14

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
```

CASE

Definition 15

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

CASE

Beispiel 16

```
CASE x
     WHEN 1, 2 THEN
          msg := 'one or two';
     ELSE
          msg := 'other value than one or two';
END CASE;
```

Simple Loop

Definition 17

```
[ <<label>> ]
LOOP
      statements
END LOOP [ label ];
```

- Endlosschleife die durch EXIT oder RETURN verlassen wird.
- Das optionale Label ist bei verschachtelten LOOPs wichtig.

Query Results durchlaufen

Beispiel 18

```
FOR order IN SELECT * FROM orders
                      ORDER BY created_at LOOP
    PERFORM send_confirmation_email(order.id)
END LOOP;
```

Intern nichts anderes als ein Cursor (Was das ist? $\rightarrow$ später)

Fehlerbehandlung

Definition 19

```
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
      ... ]
END;
```

- Standardverhalten: Auftretende Fehler brechen die Funktion ab und führen zu ROLLBACK
- Fehlerbehandlung: Über BEGIN mit EXCEPTION Block (try ... catch in Java)
- Im EXCEPTION Block: lokale Variablen bleiben erhalten, aber ROLLBACK aller Datenbankoperationen

Fehlerbehandlung

Beispiel 20

```
LOOP
    -- first try to update the key
    UPDATE db SET b = data WHERE a = key;
    IF found THEN
        RETURN;
    END IF;
    -- not there, so try to insert the key
    -- if someone else inserts the same key concurrently,
    -- we could get a unique-key failure
    BEGIN
        INSERT INTO db(a,b) VALUES (key, data);
        RETURN;
    EXCEPTION WHEN unique_violation THEN
        -- do nothing, and loop to try the UPDATE again
    END;
END LOOP;
```

Gliederung

Definition

- Cursor ermöglichen es eine Query nicht komplett auf einmal aufzurufen sondern schrittweise zu durchlaufen.
- Cursor können wie andere Variablen genutzt werden und z.B. von einer Funktion zurückgegeben werden
- Ermöglichen UPDATE/DELETE des aktuellen Datensatzes

Beispiel 21

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR
      SELECT * FROM tenk1 WHERE unique1 = key;
```

Definition 22

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

Verwendung

- Deklaration: Cursor Variable
- OPEN: Mit dem OPEN-Befehl wird das SELECT-Statement ausgewertet.
- FETCH: Einlesen des ersten bzw. des nächsten Datensatzes
- CLOSE: Schließen des Cursors (und freigeben der Ressourcen)
- Schreibzugriff mittels Cursor:
  - Beim Verarbeiten eines Datensatzes mittels Cursors erfordert die Business Logic eventuell, dass dieser Datensatz modifiziert oder gelöscht werden soll.
  - Cursor als FOR UPDATE deklarieren
  - Im UPDATE- oder DELETE-Statement kann auf den aktuellen Datensatz zugegriffen werden:
    WHERE CURRENT OF cursorref;

Beispiel

Beispiel 23

```
CREATE TABLE test (col text, num integer);
INSERT INTO test VALUES ('123', 1);
INSERT INTO test VALUES ('234', 9);

CREATE OR REPLACE FUNCTION demo() RETURNS void AS $$
DECLARE
   c CURSOR FOR SELECT * FROM test FOR UPDATE;
   c2 refcursor;
   counter integer;
   rowvar test%ROWTYPE;
BEGIN
   ... -- naechste Folie
END
$$ LANGUAGE plpgsql;

SELECT demo();
```

Beispiel

Beispiel 24

```
counter := 0;
FOR t IN c LOOP
  RAISE NOTICE '(%, %)', t.col, t.num;
  UPDATE test SET num = counter WHERE CURRENT OF c;
  counter := counter + 1;
END LOOP;

OPEN c2 FOR SELECT * FROM test WHERE num < 5;
FETCH c2 INTO rowvar;
WHILE found LOOP
   RAISE NOTICE '(%, %)', rowvar.col, rowvar.num;
   FETCH c2 INTO rowvar;
END LOOP;
CLOSE c2;

-- Ausgabe: (123, 1) (234, 9) (123, 0) (234, 1)
```

Gliederung

Einführung

- Ein Trigger weist die Datenbank an bei einem Ereignis automatisch eine (user defined) function auszuführen.
- Trigger können entweder vor (before Trigger) oder nach (after Trigger) einem INSERT, UPDATE oder DELETE, entweder einmal pro modifizierter Zeile (per row) oder einmal pro SQL Statement (per statement) ausgeführt werden.
- Die Trigger Funktion muss argumentlos sein und Rückgabewert „trigger“ haben.
- Ein Trigger wird mittels CREATE TRIGGER erstellt.

Definition 25

```
CREATE TRIGGER name { BEFORE | AFTER }
    { INSERT | UPDATE | DELETE } [OR ...]
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE funcname ()
```

Per-Row Before Trigger Funktionen

- Können NULL zurückgeben um die Operation für die jeweilige Zeile zu überspringen.
- Können durch die speziellen Variablen NEW und OLD auf den Vorher/Nachher Zustand der Zeile zugreifen.

INSERT	OLD undefiniert, NEW enthält die INSERT Werte
UPDATE	OLD und NEW definiert
DELETE	OLD enthält die „alten“ Werte, NEW undefiniert

- Bei INSERT und UPDATE Triggern ersetzt die zurückgegebene Zeile die im originalen Statement angegebene. Damit kann die Funktion die Zeile verändern.

Beispiel

Beispiel 26

```
CREATE FUNCTION t() RETURNS trigger AS $$
BEGIN
   IF NEW.empname IS NULL THEN
        RAISE EXCEPTION 'null empname';
   END IF;
   IF NEW.salary IS NULL THEN
        RAISE EXCEPTION '% null salary', NEW.empname;
   END IF;

   IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% negative salary', NEW.empname;
   END IF;

   NEW.last_date := current_timestamp;
   NEW.last_user := current_user;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Typische Trigger-Anwendungen

- Komplexe Integritätsbedingungen: Mit Triggern lassen sich wesentlich komplexere Bedingungen formulieren als mit CHECK-Constraints.
- Abgeleitete Daten: Mittels Trigger werden zusammenhängende Daten konsistent gehalten (z.B. Tabelle mit Einzelbestellungen und Spalte mit Gesamtpreis)
- ...