Prozeduren und Trigger in Datenbanken

Holger Jakobs – bibjah@bg.bib.de

2012-06-25

Prozeduren (stored procedures) bringen bei Datenbanken prinzipiell dasselbe, was sie bei gewöhnlichen Programmiersprachen auch bringen: nämlich eine einfache Möglichkeit, mehrere Anweisungen durch eine einzige Anweisung auszuführen. Dabei ist auch eine Parametrisierung möglich. Die Prozeduren werden in der Datenbank selbst hinterlegt und laufen demnach auch im Datenbanksystem, d. h. im Backend auf dem Datenbankserver. Die Sprache(n), in denen die Prozeduren geschrieben werden können, variieren je nach Datenbanksystem. Allgemein ist jedoch eine prozedurale Variante von SQL verfügbar, oft die klassische Programmiersprache C, ggf. noch weitere, z. B. Java oder Scriptsprachen.

Trigger (Auslöser) werden durch bestimmte Ereignisse abgefeuert, d. h. im Gegensatz zu einer Prozedur nicht explizit, sondern implizit, z. B. durch Einfügen, Ändern oder Löschen von Daten in der Datenbank. Einem Trigger ist entweder ein Stück Code fest zugeordnet, oder aber der Aufruf einer Prozedur. Zweck eines Triggers ist in den meisten Fällen die Sicherstellung der Datenbank-Konsistenz, wenn dies mit Hilfe der üblichen Constraints (primary key, foreign key und check) nicht möglich ist, weil die Bedingung hierfür zu komplex ist.

Mit Prozeduren und Triggern kann auch die Anwendungsprogrammierung vereinfacht werden, wenn Teile der Verarbeitungslogik in die Datenbank verlagert werden. Das hat auch den Vorteil, dass dieser Teil der Verarbeitungslogik unabhängig vom verwendeten Frontend wird.

Die Quelltexte aus diesem Dokument stehen online[1] zur Verfügung.

1) `http://www.bg.bib.de/portale/dab/Quelltexte/sources-trigger.txt`

Inhaltsverzeichnis

1 Prozeduren und Funktionen

1.1 Prozeduren und Funktionen bei Oracle

Neben den benutzerdefinierten Funktionen, die in diesem Dokument erläutert werden, gibt es eine ganze Reihe vordefinierter Funktionen, u. a. zur Konversion zwischen Datentypen, für Berechnungen, zur Zeichenkettenbearbeitung usw. Diese sind in der Originaldokumentation[1] von Oracle recht gut erläutert und werden daher in diesem Dokument ohne weitere Erklärung gelegentlich benutzt.

Bei Oracle wird zwischen Prozeduren und Funktionen unterschieden. Um Funktionen oder Prozeduren erzeugen zu dürfen, muss man das `CREATE PROCEDURE system privilege` haben. Um Funktionen oder Prozeduren innerhalb von fremden Schemas zu erzeugen, benötigt man das `CREATE ANY PROCEDURE system privilege`; um sie zu ersetzen, benötigt man `ALTER ANY PROCEDURE system privilege`. Zum Ausführen werden auch entsprechende `execute`-Rechte benötigt.

1.1.1 Syntax von PL/SQL

Die Syntax zum Erzeugen von Prozeduren und von Funktion ist sehr ähnlich:

```
CREATE [ OR REPLACE ] FUNCTION [ schema. ] function_name
  [ ( argument [ IN|OUT|IN OUT ] [ NOCOPY ] datatype [ , ... ] ) ]
  RETURN datatype AS  plsql_function_body ;

CREATE [ OR REPLACE ] PROCEDURE [ schema. ] procedure_name
  [ ( argument [ IN|OUT|IN OUT ] [ NOCOPY ] datatype [ , ... ] ) ]
  AS plsql_sql_procedure_body ;
```

Hierbei stehen `plsql_function_body` und `plsql_procedure_body` für den Rumpf der Funktion bzw. Prozedur. Statt dieses Rumpfs kann auch die Deklaration einer C- oder Java-Funktion verwendet werden – dies wird hier aber nicht weiter erläutert. Es kann zusätzlich noch angegeben werden, mit wessen Rechten die Funktion bzw. Prozedur laufen soll – auch auf diese Besonderheit wird hier nicht weiter eingegangen.

Wenn eine Prozedur oder Funktion keine Argumente hat, wird bei der Definition auch keine runde Klammer angegeben. Bei Funktionen werden die runden Klammern allerdings beim Aufruf immer angegeben, auch wenn sie ggf. leer sind.

1) `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/functions001.htm`

Die Oracle-Beispiele in diesem Dokument sind in PL/SQL geschrieben, einer prozeduralen Variante von SQL, in der man neben den Abfrage- und Cursor-Anweisungen auch Variablen deklarieren und Schleifen benutzen kann. Es gibt auch eine Ausnahmebehandlung und die Möglichkeit, Fehler an die rufende Einheit zurückzuliefern. In PL/SQL sind die üblichen Datenmanipulationskommandos erlaubt. Die PL/SQL-Programmeinheiten können verwendet werden in anonymen Blöcken, Prozeduren, Funktionen und im Anweisungsteil von Triggern (siehe Abschnitt 2.1 auf Seite 22).

1.1.2 Variablen in PL/SQL

Variablen werden in PL/SQL vor dem `BEGIN` des Anweisungsteils deklariert. Sie können jeden SQL-Datentyp haben, den Oracle kennt, oder auch den PL/SQL-spezifischen Datentyp `BOOLEAN` (der eigentlich auch zum SQL-Standard gehört, aber bei Oracle noch nicht enthalten ist). Bei der Variablendeklaration gibt man den Typen und ggf. auch die Größe an. Beispiele:

```
spielernr INTEGER;
betrag    DECIMAL(10,2);
laenge    REAL;
beginn    DATE;
genehmigt BOOLEAN;
```

Statt der hier gezeigten SQL-Standarddatentypen kann man auch die Oracle-spezifischen Typen verwenden, d. h. bei Zahlen immer `NUMBER` usw. – siehe Dokument „SQL-Datentypen und ihre Besonderheiten“[2].

Bei der Angabe des Typs für formale Parameter wird nur der Datentyp selbst verwendet, aber keine Angabe der Größe, also nur `DECIMAL` und nicht `DECIMAL(10,2)`.

Darüber hinaus besteht die Möglichkeit, zusammengesetzte Datentypen zu verwenden, d. h. Tabellen, Arrays, Records.

Die Zuweisung von Werten zu Variablen geht über den Operator `:=`, also `spielernr := 44;`. Auch die üblichen arithmetischen Berechnungen sind ohne weiteres erlaubt, wobei sowohl Literale als auch Variablen und Konstanten in den Ausdrücken verwendet werden können.

Den Wert einer Variablen kann man auch über die `INTO`-Klausel eines `SELECT`-Statements verändern oder über den Aufruf einer Prozedur, die einen Ausgabeparameter hat. Wenn man den Typ `decimal` bei Ausgabeparametern verwendet, kann es passieren, dass der Nachkommateil abgeschnitten wird – deshalb hier besser den Oracle-spezifischen Typ `number` verwenden.

```
laenge := 2.54;
SELECT max(betrag) INTO betrag FROM strafen;
```

2) https://www.bg.bib.de/portale/dab/Grundlagen/datentypen.pdf

Konstanten werden wie Variablen deklariert, aber mit dem Zusatz `CONSTANT` und einem Wert versehen.

```
grenze CONSTANT DECIMAL(10,2) := 345.40;
```

Statt einen Datentyp direkt anzugeben, kann man Variablen auch genauso deklarieren wie bereits bestehende Variablen oder Tabellenspalten. Hierzu schreibt man an Stelle des Typen den Namen der Variablen oder Tabellenspalte, gefolgt von `%TYPE`.

Eine Variable kann auch eine ganze Tabellenzeile enthalten. Sie wird als eine Variable vom selben Typ wie eine Tabelle deklariert mit `strafensatz strafen%ROWTYPE`. Um auf einzelne Spalten zuzugreifen, wird die Punkt-Notation verwendet: `nr := strafensatz.spielnr;`

Bei `FETCH`-Anweisungen wird eine Zeile in den Strafensatz geladen: `FETCH c1 INTO strafensatz;` (wobei `c1` ein Cursor sein muss, der einen passenden Satz liefert).

1.1.3 Cursor in PL/SQL

Innerhalb von PL/SQL kann man mit Cursorn ähnlich arbeiten wie bei Embedded SQL in C und bei JDBC. In diesem Zusammenhang werden dann auch die Anweisungen `OPEN`, `FETCH` und `CLOSE` benutzt. Cursor werden ähnlich wie Variablen deklariert. Bei der Deklaration kann bereits eine Abfrage festgelegt werden, aber generischen Cursorn wird die Abfrage erst beim Öffnen zugewiesen.

```
create or replace procedure curstest1 as
  cursor curs1 is
    select spielname, ort
    from sp_spieler where ort='Bonn';
  name   varchar(20);
  stadt  varchar(20);
begin
  open curs1;
  fetch curs1 into name, stadt;
  if curs1%found then
    insert into tabelle1 values (name, stadt);
  end if;
  close curs1;
end;
```

Diese kleine Prozedur enthält einen expliziten Cursor mit einer fest vorgegebenen Abfrage. Der Cursor wird geöffnet. Nach einem fetch-Versuch wird geprüft, ob es erfolgreich war. Wenn ja, werden Daten in eine andere Tabelle eingefügt. Am Ende wird der Cursor geschlossen.

Tabelle 1.1 auf der nächsten Seite zeigt, welche Informationen über einen Cursor abgerufen werden können. Es wird der Cursorname, direkt gefolgt von einem Prozentzeichen und

Tabelle 1.1: Informationsmöglichkeit über Cursor

Zeitpunkt	`%notfound`	`%found`	`%rowcount`	`%isopen`
vor `open`	Fehler	Fehler	Fehler	false
direkt nach `open`	null	null	0	true
nach erfolgreichem `fetch`	false	null	Zahl ≥ 1	true
nach erfolglosem `fetch`	true	false	Zahl ≥ 0	true
nach `close`	Fehler	Fehler	Fehler	false

einer der genannten Zeichenketten geschrieben, siehe Beispiel im Code oben. In den mit „Fehler“ markierten Situationen wird die Exception `invalid_cursor` geworfen (Exceptions siehe Tabelle 1.2 auf Seite 9).

```
create or replace procedure curstest2 as
  type cursortyp1 is ref cursor;
  curs1 cursortyp1;
  name   varchar(20);
  stadt  varchar(20);
begin
  open curs1 for select spielname, ort
    from sp_spieler where ort='Duesseldorf';
  fetch curs1 into name, stadt;
  if curs1%found then
    insert into tabelle1 values (name, stadt);
  end if;
  close curs1;
end;
```

Dieses Beispiel ist von der Funktion her wie das obige, nur dass der Cursor hier als Variable deklariert wurde. Hierzu ist es notwendig, zunächst einen Datentypen zu deklarieren. Beim Öffnen der Cursor-Variablen wird dann die Abfrage mitgegeben. Auf diese Weise kann man dieselbe Cursor-Variable für verschiedene Abfragen benutzen.

Eine Alternative zu expliziten `open`-, `fetch`- und `close`-Anweisungen ist die Verwendung von `LOOP`. In einem Beispiel sollen hier Angaben aus der Strafentabelle des Sportvereins verarbeitet werden. Ziel ist es, in eine neue Tabelle die Strafen eines Spielers einzutragen, wobei aber nicht die Einzelbeträge, sondern die insgesamt aufgelaufenen Beträge enthalten sind. Dies geschieht innerhalb einer Schleife in der Prozedur.

```
create table str (
```

```
  znr   integer primary key,
  spnr  integer not null,
  datum date not null,
  sum   decimal(10,2) not null
);

create procedure p1 (spnr integer) as
  cursor c1 is select zahlnr, spielnr, datum, betrag
  from sp_strafen where spielnr = spnr order by datum;
  betragsum decimal(10,2);
begin
  betragsum := 0;
  delete from str;
  for satz in c1 loop
    betragsum := betragsum + satz.betrag;
    insert into str
      values (satz.zahlnr, satz.spielnr, satz.datum, betragsum);
  end loop;
end;
```

Nach dem Ausführen der Prozedur mit dem Parameter 44 sind alle Strafen des Spielers 44 in der Tabelle `str` enthalten. Die Ausführung geschieht interaktiv mit einem anonymen Block:

```
begin
  p1 (44);
end;
```

1.1.4 Parametrisierte Cursor in PL/SQL

Cursor können auch parametrisiert sein, so dass Werte z. B. in `where`-Bedingungen zur Laufzeit ausgetauscht werden können.

```
create or replace procedure curstest3 (spielort in varchar) as
  cursor curs1 (wohnort in varchar) is
    select spielname, ort
    from sp_spieler where ort = wohnort;
  name   varchar(20);
  stadt  varchar(20);
begin
  open curs1 (spielort);
  fetch curs1 into name, stadt;
  if curs1%found then
```

```
    insert into tabelle1 values (name, stadt);
  end if;
  close curs1;
end;
```

In diesem Beispiel wird dem Prozedurparameter `spielort` ein Wert übergeben, der wiederum an den parametrisierten Cursor weitergegeben wird. Der Rest funktioniert wie gehabt.

1.1.5 Ablaufsteuerung in PL/SQL

Als prozedurale Variante von SQL gibt es in PL/SQL alle üblichen Strukturen zur Ablaufsteuerung, d. h. Verzweigungen und Schleifen.

Verzweigung und Schleifen

```
if bedingung then
  ... -- Anweisungen
elsif bedingung then
  ... -- Anweisungen
else
  ... -- Anweisungen
end if;

loop
  ... -- Anweisungen
  exit when bedingung
  ... -- Anweisungen
end loop;

for i in [ reverse ] klein..gross loop
  ... -- Anweisungen
end loop;

while gehalt < 1000 loop
  ... -- Anweisungen
end loop;
```

Die numerische `for`-Schleife kann vorwärts oder rückwärts ablaufen, aber die untere Grenze steht immer zuerst, auch wenn `reverse` angegeben ist.

Die Schleifen können durch die Anweisung `exit` verlassen werden. Dies kann in einem `if`-Konstrukt geschehen oder mittels `exit when bedingung`.

Die Existenz des unstrukturierten `goto` soll hier verschwiegen werden. Hilfreich beim Programmieren ist allerdings die Ausnahmebehandlung. Hierzu schreibt man eine `exception`-Klausel am Ende des `begin end`-Blocks. Ausnahmen können vom Datenbanksystem

Tabelle 1.2: Einige der vordefinierten Ausnahmen bei Oracle

Exception	Code	Erläuterung
DUP_VAL_ON_INDEX	ORA-00001	uneindeutiger Wert in einem eindeutigen Index (oder Primärschlüssel)
TIMEOUT_ON_RESOURCE	ORA-00051	Zeitüberschreitung beim Warten, evtl. Deadlock
INVALID_CURSOR	ORA-01001	ungültiger Cursor, Definition vergessen?
NOT_LOGGED_ON	ORA-01012	Anmeldung vergessen?
LOGIN_DENIED	ORA-01017	Einloggen fehlgeschlagen, Kennwort falsch?
NO_DATA_FOUND	ORA-01403	keine Daten gefunden (wie sqlcode=100)
ZERO_DIVIDE	ORA-01476	Division durch Null
ROWTYPE_MISMATCH	ORA-06502	Datentyp des Abfrageergebnisses und der Hostvariablen stimmen nicht überein.
CURSOR_ALREADY_OPEN	ORA-06511	Versuch, einen bereits geöffneten Cursor zu öffnen
OTHERS		alle anderen Fehler; kann benutzt werden mit einer WHEN OTHERS-Klausel

selbst kommen, z. B. die Ausnahme no_data_found oder aber mit Hilfe einer raise-Anweisung vom selbst geschriebenen Code generiert werden.

Mit Hilfe der Anweisung raise_application_error (nr, 'text'); erzeugt man einen Anwendungsfehler, der an das aufrufende Programm weitergegeben wird, also nicht innerhalb der Prozedur selbst behandelt wird. Die angegebene nr muss sich im Bereich −20000 bis −20999 bewegen und wird als sqlcode zurückgeliefert, der angegebene text in der SQL-Message.

Exception Handling

```
create function f2 (spnr in integer) return varchar as
  titel varchar(5);
  fehlt exception;
begin
  select titel into titel from spieler where spielnr = spnr;
  if titel is null then
    raise fehlt;
  else
    return titel;
  end if;
exception
  when fehlt then return '(keiner)';
```

```
    when no_data_found then
      raise_application_error(-20100, 'Spieler nicht da');
    when others then
      raise_application_error(-20123, SQLERRM);
  end;
```

Fängt man die erzeugte Ausnahme nicht ab, so wird ein Fehler geliefert (`unhandled user-defined exception`).

Neben den benutzerdefinierten Ausnahmen gibt es auch eine ganze Reihe von Oracle vordefinierte Exceptions, siehe Tabelle 1.2 auf der vorherigen Seite. Diese führen nicht zu einem Anwendungsfehler, falls man sie nicht abfängt. Im obigen Beispiel wird die Ausnahme `no_data_found` geworfen, wenn es den Spieler mit der Nummer `spnr` nicht gibt. Wir erzeugen dann einen Anwendungsfehler, der auch beim Aufrufen über ein interaktives Frontend (z. B. TOra) auftaucht. Beim Auftreten unerwarteter Exceptions wird die Original-Oracle-Meldung (`sqlerrm`) mit der Fehlernummer -20123 geliefert. `sqlcode` und `sqlerrm` stehen immer als Variablen zur Verfügung.

Innerhalb einer Prozedur oder Funktion kann es auch mehrere Blöcke geben, in denen Exceptions abgefangen werden. Hier ein kleines Beispiel einer Prozedur, die das Datum des letzten Kundenkontakts in der Tabelle `kunde` pflegen soll. Ist es der erste Kontakt im aktuellen Kalenderjahr, so wird das bislang letzte Kontaktdatum in einer History-Tabelle eingetragen, in der der jeweils letzte Kontakt eines Kunden für die vergangenen Jahre steht.

```
create or replace procedure p2 (kundennr in integer) as
  datum date;
  letztesJahr integer;
begin
  begin
    select letzterKontakt into datum
        from kunde where kdnr = kundennr;
    if datum < trunc(sysdate, 'YEAR') then
      select to_number(to_char(sysdate, 'YYYY')) - 1
          into letztesJahr from dual;
      insert into Kontakt values (kundennr, letztesJahr, datum);
    end if;
  exception
    when no_data_found then
      raise_application_error(-20100, 'Kunde ' || kundennr ||' nicht da');
  end;
  update kunde set letzterKontakt = trunc(sysdate)
      where kdnr = kundennr;
end;
```

Leider gibt es bei Oracle keine direkte Möglichkeit, nach einer Datenmanipulationsanweisung festzustellen, ob und wenn ja wie viele Tupel von ihr betroffen wurden. Ein `SELECT`

auf dieselbe Tabelle mit derselben `where`-Klausel muss als umständliches Vehikel dafür angesehen werden.

1.1.6 Weitere Möglichkeiten in PL/SQL

Auf die weiteren Möglichkeiten von PL/SQL, d. h. Object Types, tabellenwertige Variablen und Collections geht dieses Dokument nicht ein.

1.1.7 Einschränkungen bei Funktionen

Benutzerdefinierte Funktionen können nicht verwendet werden

- in einem `check` Constraint
- in `default`-Klauseln bei der Tabellendefinition

Wenn Funktionen innerhalb einer Abfrage oder einer Datenmanipulationsanweisung verwendet werden, dürfen sie

- keine `OUT`- oder `IN OUT`-Parameter haben.
- kein `COMMIT` oder `ROLLBACK` ausführen, auch nicht implizit durch Datendefinitionskommandos.
- keine Daten schreiben, wenn sie von einer Abfrage aufgerufen werden. Wohl aber geht das, wenn sie von einer Datenmanipulationsanweisung aus aufgerufen werden. Verwenden Sie `insert into x values(funk1(100));` an Stelle von `select funk1(100) from dual;`, damit die Funktion `funk1()` innerhalb einer DML-Anweisung und nicht innerhalb einer Abfrage aufgerufen wird. `x` ist eine Testtabelle, die den Rückgabewert von `funk1()` aufnimmt, so dass Sie ihn anschließend sichtbar machen können.
- nicht auf dieselbe Tabelle schreiben wie die Datenmanipulationsanweisung, die sie aufruft.

1.1.8 Weitere Beispiele zu Oracle-Prozeduren und -Funktionen

```
create function sum_strafen (spielernr in integer)
return decimal as
  summe decimal(10,2);
begin
  select sum(betrag) into summe
  from strafen
  where spielnr = spielernr;
  return (summe);
end;
```

```
select sum_strafen(44) from dual;   -- Testaufruf

create procedure strafen_betrag (znr in integer,
  neubetrag in decimal) as
begin
  update strafen
  set betrag = neubetrag
  where zahlnr = znr;
end;
begin -- anonymer Block
  strafen_betrag (1, 35.00);  -- Testaufruf
end;
```

Die `select`-Abfrage nach dem Wert der Funktion `sum_strafen` bezieht sich auf die Pseudotabelle `dual`, weil bei Oracle eine Abfrage unbedingt eine `from`-Klausel mit Tabelle haben muss (vgl. `select sysdate from dual;`).

Die Prozedur `strafen_betrag` wird nach ihrer Definition in einem anonymen `begin end`-Block ausgeführt. Ein solcher Block ist für den Aufruf notwendig; lediglich bei Verwendung des SQL*Plus-Frontends von Oracle kann man statt dessen auch `execute strafen_betrag (1, 35.00)` schreiben.

1.1.9 Aufruf von Prozeduren aus Programmen

Embedded SQL in C

Innerhalb von Embedded SQL schreibt man den Aufruf einer Prozedur wie folgt. `zahlnr` und `betrag` müssen hier als Hostvariablen vereinbart sein, die innerhalb von SQL-Statements mit einem führenden Doppelpunkt angesprochen werden.

```
exec sql execute
  call strafen_betrag (:zahlnr, :betrag);
```

Die Prozedur übernimmt die Parameter in der angegebenen Reihenfolge – wie bei anderen Programmiersprachen auch. Allerdings kann man von der Reihenfolge abweichen, wenn man die Parameter benennt, hier beispielsweise so:

```
exec sql execute
  call strafen_betrag (neubetrag=>35.00, znr=>1);
```

Natürlich muss wie immer eine Fehlerbehandlung durchgeführt werden, die hier aber nicht gezeigt ist.

Java mit JDBC

In Java-JDBC-Programmen werden Prozeduren so aufgerufen, wobei `conn` die bestehende Verbindung zur Datenbank ist:

```
CallableStatement callstmt = conn.prepareCall ("{call strafen_betrag (?, ?)}");
callstmt.setInt ("znr", 1);
callstmt.setDouble ("neubetrag", 35.00);
callstmt.execute();
```

Statt der geschweiften Klammern können auch `begin` und `end` verwendet werden, was dann so aussieht: `conn.prepareCall ("begin call strafen_betrag (?, ?); end;");`

In jedem Fall müssen die Statements als Variable vom Typ `CallableStatement` deklariert und „vorbereitet“ (prepared) werden. Danach werden die einzelnen Parameter mit Werten versehen und dann das Statement ausgeführt. Natürlich muss wie immer eine Fehlerbehandlung durchgeführt werden, die hier aber nicht gezeigt ist.

1.1.10 Prozeduren und Funktionen im Vergleich

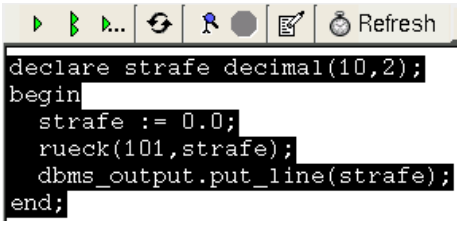
Funktionen werden immer als Bestandteil eines Ausdrucks aufgerufen, während Prozeduraufrufe eigenständige Anweisungen sind. Daher können Prozeduren nicht direkt aus SQL-Anweisungen aufgerufen werden. Allerdings können Funktionen, die Bestandteil eines Ausdrucks sind, ihrerseits wiederum Prozeduren aufrufen.

Bei Funktionen werden hinter dem Namen immer runde Klammern verwendet. Bei parameterlosen Funktionen sind sie leer, ansonsten stehen dort die Argumente in derselben Reihenfolge wie in der Funktionsdefinition. Benannte Argumente wie bei Prozeduren sind nicht möglich.

1.1.11 Hinweise zur Fehlersuche

Wenn man Funktionen oder Prozeduren eingerichtet hat, werden manche Fehler sofort gemeldet, manche aber auch nicht oder zum Ausführungszeitpunkt. Falls keine klare Fehlermeldung vom Frontend angezeigt wird, kann man evtl. mit `select * from user_errors;` ausführlichere Information über den Fehler bekommen. Dort werden auch Zeilen- und Spaltennummer aufgeführt, wo der Fehler aufgetreten ist.

Diese Art der Fehlermeldung ist vom Frontend unabhängig, weil sie komplett auf dem Server verwaltet wird.



Eine weitere Möglichkeit, Werte sichtbar zu machen, besteht in der Verwendung von `dbms_output.put_line(...)`, wobei statt der Punkte Zeichenketten oder Variableninhalte ausgegeben werden können. Um die Ausgabe sichtbar zu machen, muss in SQLplus `set serveroutput on;` eingegeben werden. In TOra wird unter „Tools“ der „SQL Output“ geöffnet. Um in TOra Ausgabeparameter von Prozeduren

verwenden zu können, müssen diese deklariert werden. Die Deklaration und der anonyme Block werden *gemeinsam* markiert und dann durch Klicken auf den grünen Pfeil ausgeführt – siehe nebenstehende Abbildung. Das SQL-Output-Fenster in TOra muss manuell aktualisiert werden; ansonsten erscheinen die Ausgaben verzögert.

1.2 Funktionen bei PostgreSQL

Bei PostgreSQL wird nicht streng zwischen Funktionen und Prozeduren unterschieden, es ist einfach so, dass immer die Anweisung `CREATE FUNCTION` verwendet wird, der Rückgabewert aber wahlfrei ist. Der Funktionscode wird als Textstring übergeben. Zusätzlich muss man noch angeben, in welcher Sprache die Prozedur geschrieben wurde, denn es kann mehrere Interpreter innerhalb der Datenbank geben. Bei uns ist auf jeden Fall die PostgreSQL-Variante von PL/SQL namens `plpgsql` vorhanden, im Folgenden PL/pgSQL genannt.

1.2.1 Syntax von PL/pgSQL

Die Syntax zum Erzeugen von Funktionen in PL/pgSQL lautet:

```
create [ or replace ] function function_name
  ( datatype [ , ... ] )
  returns datatype AS  '
    plpgsql_function_body
  ' language 'plpgsql' ;
```

Hierbei steht `plpgsql_function_body` für den Rumpf der Funktion in Form eines Textstrings – daher auch die Hochkommas drumherum. Statt dieses Rumpfs kann auch die Deklaration einer C-Funktion verwendet werden – dies wird hier aber nicht weiter erläutert.

Der Rückgabetyp hinter `returns` ist zwingend erforderlich und stellt einen SQL-Datentyp dar oder lautet `void`, `record`, `trigger`[3] oder `setof datentyp` – letzteres wird hier nicht weiter erläutert. Der Ablauf einer Funktion wird durch eine `return`-Anweisung beendet; sie ist zwingend erforderlich.

Die Parameterliste ist immer vorhanden, d. h. die runden Klammern sind ggf. leer, falls keine Parameter verwendet werden. Parameter können Eingabeparameter (`IN`, ohnehin default), Ausgabeparameter (`OUT`) oder Ein-/Ausgabeparameter (`INOUT`) sein. Nur Ein- und Ein-/Ausgabeparameter werden beim Funktionsaufruf mitgegeben. Funktionen können in `SELECT`-Anweisungen hinter `FROM` wie Tabellen angegeben werden.

Die PostgreSQL-Beispiele in diesem Dokument sind in PL/pgSQL geschrieben, einer prozeduralen Variante von SQL, in der man neben den Abfrage- und Cursor-Anweisungen auch Variablen deklarieren und Schleifen benutzen kann. Es gibt auch eine Ausnahmebehandlung und die Möglichkeit, Fehler an die rufende Einheit zurückzuliefern. In PL/pgSQL sind

3) siehe Abschnitte 1.2.8 auf Seite 20 und 2.2 auf Seite 27

die üblichen Datenmanipulationskommandos erlaubt. Um festzustellen, auf wie viele Tupel das letzte Kommando einen Effekt hatte, kann man `GET DIAGNOSTICS` *`integervariable`* `= ROW_COUNT` verwenden. Außerdem setzen alle Kommandos einschließlich `SELECT` die boolesche Variable `FOUND` auf `true`, sofern mindestens 1 Tupel betroffen ist.

Ein Funktionsrumpf stellt einen Anweisungsblock dar. Ein Anweisungsblock besteht aus einem wahlfreien `declare`-Teil, in dem Variablen deklariert werden können, und einem Anweisungsteil, der von `begin` und `end` eingerahmt wird. Kommentare können in der in SQL üblichen Variante mit zwei Minuszeichen oder in der C-Variante mit `/* ... */` eingefügt werden.

Funktionen in PL/pgSQL werden immer *innerhalb* einer Transaktion ausgeführt, daher können die Kommandos `begin transaction`, `commit` und `rollback` nicht innerhalb der Funktionen vorkommen.

Gelöscht werden können Funktionen mittels `drop function function_name (typ, ...)`. Da Funktionen überladen werden können, ist die Angabe der Parametertypliste notwendig. Durch Zusatz von `cascade` würden auch die Trigger, die sie verwenden, automatisch mit gelöscht. Andernfalls müssten die Trigger vorher gelöscht werden, sofern vorhanden.

1.2.2 Variablen in PL/pgSQL

Variablen werden im `declare`-Teil vereinbart. Sie können alle SQL-Datentypen haben, oder auch den Typ einer Tabellenspalte oder gar einer Tabellenzeile.

```
declare
  spielnr   integer;
  zeile     spieler%rowtype;
  satz      record;
  plz       spieler.plz%type;
```

Ob man bei `zeile` den Zusatz `%rowtype` hinzufügt oder weglässt, ist ohne Bedeutung, da jede Tabelle automatisch einen gleichnamigen Rowtyp erzeugt. Die Variante mit `%rowtype` ist der Schreibweise von Oracle ähnlicher. Wahlweise kann man auch einfach `record` schreiben, was ein Platzhalter-Datentyp ist.

Die allgemeine Syntax für eine Variablendeklaration lautet
`name` `[ constant]` *`datentyp`* `[ not null ] [default | := ausdruck]`;

Durch Initialisierung mittels `:=` oder `default` wird die Variable beim Betreten des Blocks mit einem Wert versehen. Durch den Zusatz `constant` wird eine Konstante statt einer Variablen vereinbart, die natürlich einen Wert bekommen muss, ebenso wie eine Variable, die als `not null` deklariert wird. Die Initialisierung findet jedes Mal beim Betreten des Blocks statt.

Die Parameter einer Funktion haben zunächst keinen Namen, sondern sind durchnumeriert mit `$1`, `$2` usw. Möchte man lieber benannte Parameter haben, so deklariert man lokale Variablen als Alias für sie.

```
create function sum_strafen (integer) returns decimal as '
  declare
    spielernr alias for $1;
    summe decimal(10,2);
  begin
    select sum(betrag) into summe
    from strafen
    where spielnr = spielernr;
    return summe;
  end;
' language 'plpgsql';
```

Parametertypen und auch Rückgabetyp einer Funktion können auch unbestimmt sein, so dass sie erst zur Laufzeit festgelegt werden. Auf diese Weise kann man polymorphe Funktionen schreiben. Ohne wirklichen Datenbankbezug, aber ein einfaches Beispiel dazu wäre folgendes. Der Funktion kann man alle Argumente übergeben, die zueinander kompatibel und mittels des +-Operators verknüpft werden können.

```
create function add (anyelement, anyelement) returns anyelement as '
  begin
    return $1 + $2;
  end;
' language 'plpgsql';
```

Zuweisungen geschehen einfach durch den Operator `:=`, wobei auf der rechten Seite ein gültiger Ausdruck stehen muss, der zum Datentyp der Variablen auf der linken Seite zuweisungskompatibel ist. Auch durch ein `select ... into ...` kann eine Variable einen neuen Wert bekommen.

```
  laenge := 2.54;
  select max(betrag) into betrag from strafen;
```

Sollte bei `select` kein passendes Tupel gefunden worden sein, so wird der Wert `null` geliefert. Das kann man ggf. abfangen und durch einen anderen Wert ersetzen:

```
  select max(betrag) into betrag from strafen;
  if not found then
    betrag := -1;
  end if;
```

1.2.3 Aufruf von Funktionen in PL/pgSQL

Funktionen können überall dort eingesetzt werden, wo ein Ausdruck gefordert wird. Liefert eine Funktion aber gar keinen Wert zurück, sondern hat nur einen nützlichen Nebeneffekt,

kann man sie mittels `perform` aufrufen. Bei der Angabe von Anführungszeichen ist darauf zu achten, dass diese doppelt gesetzt werden müssen, falls man sich in einem PL/pgSQL-Block befindet, weil ansonsten der Quelltext für diesen Block beendet würde.

```
perform f2 (''Meier'', 17);
```

1.2.4 Dynamisches Ausführen von Code in PL/pgSQL

Um PL/pgSQL-Code auszuführen, den man in einer Variablen vom Typ `text` zusammengebaut hat, verwendet man das Kommando `execute`. Innerhalb des dahinter angegebenen Kommandostrings findet keinerlei Variablenersetzung statt, so dass man die passenden Werte beim Konstruieren des Strings einbauen muss.

Ergebnisse von `select`-Anweisungen innerhalb des Strings überleben das Ende von `execute` nicht. Leider ist die Verwendung von `select into` hier nicht erlaubt. Um Ergebnisse aus einem dynamisch erzeugten SQL-Kommando zu erhalten, muss man eine Schleife mit `for`, `in` und `execute` oder einen Cursor mit `open`, `for` und `execute` benutzen, siehe Abschnitt 1.2.6 auf der nächsten Seite.

1.2.5 Ablaufsteuerung in PL/pgSQL

Als prozedurale Variante von SQL gibt es in PL/pgSQL alle üblichen Strukturen zur Ablaufsteuerung, d. h. Verzweigungen und Schleifen.

```
if bedingung then
  ... -- Anweisungen
elsif bedingung then
  ... -- Anweisungen
else
  ... -- Anweisungen
end if;

loop
  ... -- Anweisungen
  exit when bedingung
  ... -- Anweisungen
end loop;

for i in [ reverse ] klein..gross loop
  ... -- Anweisungen
end loop;

for i in selectanweisung loop
  ... -- Anweisungen
```

```
  end loop;

  while gehalt < 1000 loop
    ... -- Anweisungen
  end loop;
```

Die numerische `for`-Schleife kann vorwärts oder rückwärts ablaufen, aber die untere Grenze steht immer zuerst, auch wenn `reverse` angegeben ist.

Die Schleifen können durch die Anweisung `exit` verlassen werden. Dies kann in einem `if`-Konstrukt geschehen oder mittels `exit when bedingung`.

1.2.6 Cursor in PL/pgSQL

Cursor dienen der sequentiellen Verarbeitung von Abfrageergebnissen.. Sie können gebunden sein, ungebunden oder parametrisiert, so dass man insgesamt drei verschiedene Arten der Cursordeklaration verwenden kann:

```
declare
  cursor1 refcursor;
  cursor2 cursor for select * from strafen;
  cursor3 cursor (nr integer) is
    select * from strafen where spielnr=nr;
```

`cursor1` kann mit jeder beliebigen Abfrage verwendet werden, die dann beim Öffnen des Cursors angegeben werden muss. `cursor2` ist ein gebundener Cursor, weil die Abfrage schon feststeht, während `cursor3` ein parametrisierter Cursor ist, dem man beim Öffnen nur noch die passenden Werte für die Parameter mitgibt. So könnte man sie öffnen:

```
open cursor1 for select * from teams where spielnr = nummer;
```

Beim Öffnen von `cursor1` muss `nummer` eine PL/pgSQL-Variable von einem zu `spielnr` passenden Typ sein. Sie wird durch ihren aktuellen Wert ersetzt.

```
open cursor1 for execute ''select * from '' || quote_ident(param1);
```

`param1` muss jetzt ein benannter Parameter oder eine andere Variable in der aktuellen Funktion sein; wenn der Parameter unbenannt ist, tut es natürlich auch ein `$n` für den n-ten Parameter. Die Funktion `quote_ident()` sorgt dafür, dass der Name richtig übergeben wird.

Einen gebundenen Cursor öffnet man einfach durch Angabe des Namens; wenn er parametrisiert ist, gibt man die Parameter in Klammern dahinter an:

```
open cursor2;
open cursor3 (44);
```

Nachdem die Cursor geöffnet wurden – bitte darauf achten, keine Cursor zu öffnen, die schon offen sind –, kann man mittels `fetch` die Ergebnistupel abholen, wobei die hinter `into` angegebenen Variablen vom Typ und von der Anzahl her zur Abfrage passen müssen, oder es wird ein passender Rowtyp verwendet. Anschließend muss der Cursor wieder geschlossen werden. Bei `fetch` und `close` unterscheiden sich die drei Cursorarten nicht.

```
fetch cursor1 into tnr, snr, liga;
close cursor1;
```

Cursor können auch als Funktionswert an die rufende Einheit zurückgeliefert werden. Das ist sinnvoll, um mehrere Tupel oder Spalten zurückzugeben, gerade bei großen Ergebnismengen. Wie bereits oben bei der Deklaration des ungebundenen Cursors gesehen ist der passende Datentyp hierfür `refcursor`. Hier ein Beispiel, das zeigt, wie man einen Cursornamen an eine Funktion übergeben kann:

```
create function getcursor (refcursor) returns refcursor AS '
  declare
    cursor_name alias for $1;
  begin
    open cursor_name for select teamnr from teams;
    return cursor_name;
  END;
' language 'plpgsql';

begin transaction;
select getcursor('beispielcursor');
fetch all in beispielcursor;
commit transaction;
```

Bei der Ausführung der Funktion `getcursor()` ist darauf zu achten, dass dies innerhalb einer Transaktion geschieht (also nicht mit autocommit), weil bei Transaktionsende der Cursor automatisch geschlossen würde. Dann liefert das `fetch` die Meldung `ERROR: cursor "beispielcursor" does not exist`.

Eine einfache Möglichkeit, Werte eines Abfrageergebnisses zu verarbeiten, ist die zweite Variante der `for`-Schleife.

```
create table str (
  znr   integer primary key,
  spnr  integer not null,
  datum date not null,
  sum   decimal(10,2) not null
);

create function s1 (integer) returns void as '
```

```
declare
  satz record;
  spnr alias for $1;
  betragsum decimal := 0;
begin
  delete from str;
  for satz in
    select zahlnr, spielnr, datum, betrag
    from strafen where spielnr = spnr order by datum
  loop
    betragsum := betragsum + satz.betrag;
    insert into str
      values (satz.zahlnr, satz.spielnr, satz.datum, betragsum);
  end loop;
  return;
end;
' language 'plpgsql';
```

Nach dem Ausführen der Prozedur mit dem Parameter 44 sind alle Strafen des Spielers 44 in der Tabelle `str` enthalten. Die Deklaration eines Cursors ist hier nicht einmal erforderlich, denn das geschieht automatisch intern.

1.2.7 Fehlerbehandlung in PL/pgSQL

PL/pgSQL hat keine umfangreichen Möglichkeiten der Fehlerbehandlung. Fehler führen immer dazu, dass eine Funktion abgebrochen und damit auch die Transaktion zurückgefahren wird. Meldungen können vom Datenbanksystem erzeugt werden oder auch von selbst geschriebenen Anweisungen. Es wird zwischen verschiedenen Stufen (level) unterschieden, ob es nur ein Hinweis oder ein schwerer Fehler ist. Folgende Stufen gibt es: `debug`, `log`, `info`, `notice`, `warning`, `exception`. Nur `exception` führt zum Abbruch der Funktion und damit der Transaktion, die anderen erscheinen z. B. in der Log-Datei des Datenbankservers.

Die Aufrufsyntax sieht so aus: `raise` *`level 'format' [, variable ...];`*

Innerhalb der Formatangabe werden Prozentzeichen durch den Inhalt der entsprechenden Variablen ersetzt – ganz analog zu `printf()` in C, aber ohne Angabe von irgendwelchen Typ- oder Längenangaben. Beispiel:

```
raise exception 'Fehler: Wert (%) nicht eindeutig', spielnr;
```

1.2.8 Triggerfunktionen

Funktionen, die für Trigger (siehe Abschnitt 2.2 auf Seite 27) benutzt werden, haben keine Parameter und den Rückgabetyp `trigger`. Eine Triggerfunktion für das Prüfen der Zimmerbelegung in der Krankenhaus-Datenbank sieht wie folgt aus:

```
  create function freibetten () returns trigger AS '
  declare
    anzfrei integer;
  begin
    select betten - count(*) into anzfrei
    from zimmer natural join patient
    where zimmer.znr=new.znr
    group by zimmer.znr, betten;
    if anzfrei < 0 then
      raise exception ''% für Tabelle "%": Zimmer % würde überbelegt'',
          TG_OP, TG_RELNAME, new.znr;
    end if;
    return new;
  END;
' language 'plpgsql';
```

Zunächst wird eine Variable `anzfrei` deklariert. Es können hier alle in PostgreSQL zulässigen Datentypen verwendet werden. Zwischen `begin` und `end` steht der auszuführende Code, der mit einem `select`-Statement beginnt. Wegen der `into`-Klausel sieht es fast aus wie ein Embedded SQL Kommando, allerdings hat die Variable `anzfrei` hinter dem `into` keinen führenden Doppelpunkt, denn es ist eine SQL-Variable und keine aus einer anderen Sprache, die in SQL eingebettet wurde.

In der Verzweigung mit `if` wird das Ergebnis der Abfrage mit 0 verglichen. Für den Fall, dass das Zimmer also überbelegt ist, wird mittels `raise exception` ein Fehler erzeugt, der dann an das Anwendungsprogramm (oder auch an `psql`) weitergegeben wird. Die Fehlermeldung muss in doppelt geschriebenen einfachen Hochkommas stehen. Das Doppeltsetzen ist notwendig, weil der gesamte Code schon in Hochkommas steht. In den Fehlertext werden drei Systemvariablenwerte eingefügt: `TG_OP` ist die aktuell ausgeführte Operation (`INSERT`, `UPDATE`, `DELETE`), `TG_RELNAME` ist der Name der Relation (Tabelle), `new` bezeichnet allgemein das neu eingefügte Tupel (bei einem `UPDATE` gibt es auch `old`, bei `DELETE` gibt es nur `old`), wir greifen hier auf das Attribut `znr` zu.

Andernfalls wird kein Fehler erzeugt, so dass die Anweisung, die den Trigger abgefeuert hat, wirksam durchgeführt wird. Nach Abschluss des Hochkommas für den Programmcode muss noch die Sprache angegeben werden.

Näheres zu Triggern in PostgreSQL siehe Abschnitt 2.2 auf Seite 27.

2 Trigger

Trigger sind ein vielfältiges Mittel zur Absicherung der Konsistenz bei komplexeren Prüfungen als es die einfachen Constraints vermögen. Beispiel in unserer Krankenhaus-Datenbank ist beispielsweise die Absicherung, dass einem Zimmer nicht mehr Patienten zugewiesen werden können als das Zimmer Betten hat. Dieses Beispiel soll im Folgenden auch verwendet werden.

Trigger können definiert sein, um vor (`before`) oder nach (`after`) dem Abarbeiten der gesamten Anweisung (`statement`) oder vor oder nach dem Verarbeiten jedes einzelnen Tupels (`row`) abgefeuert zu werden. Es gibt also insgesamt vier verschiedene Möglichkeiten, den Abfeuerungszeitpunkt zu bestimmen:

- `before statement` – wird 1x pro Anweisung ausgeführt, und zwar ganz am Anfang, auch wenn diese Anweisung viele Tupel betrifft oder auch gar keins.
- `before row` – wird 1x pro Tupel ausgeführt, und zwar vor der eigentlichen Datenbankoperation.
- `after row` – wird 1x pro Tupel ausgeführt, und zwar nach der eigentlichen Datenbankoperation.
- `after statement` – wird 1x pro Anweisung ausgeführt, und zwar ganz am Ende, nachdem alle Tupel verarbeitet wurden (auch wenn es keine waren).

2.1 Trigger bei Oracle

2.1.1 Erzeugen von Triggern in Oracle

Bei Oracle umfasst die Erzeugung eines Triggers gleich den zugehörigen Anweisungsblock. Prinzipiell sieht die Erzeugung eines Triggers daher fast so aus wie die Erzeugung einer Prozedur. Es ist nicht notwendig, den Codeblock als Prozedur abzulegen.

Die Syntax zum Erzeugen eines Triggers lautet:

```
create [ or replace ] trigger triggername
{ after | before }
{ insert | update | delete [ or { insert | update | delete } ... ] }
ON tabellenname
[ for each row [ when bedingung ] ]
[ declare deklarationsteil ]
```

```
begin
  anweisungen
[ exception ausnahmebehandlungsteil ]
end [ triggername ] ;
/
```

Triggernamen müssen innerhalb eines Schemas eindeutig sein, d. h. auch wenn sie sich auf verschiedene Tabellen beziehen, dürfen sie nicht den selben Namen tragen. Wenn man Trigger in einem interaktiven Tool eingibt, muss hinter dem `end;` noch ein Slash (`/`) allein in der nächsten Zeile stehen – andernfalls erhält man die unspezifische Meldung „`success with compilation error`“.

Trigger können vor oder nach der Verarbeitung der gesamten Anweisung (sogenannte Anweisungstrigger) oder vor oder nach Verarbeitung jedes einzelnen Tupels (sogenannte Zeilentrigger) abgefeuert werden. Bei Zeilentriggern wird der Zusatz `for each row`) verwendet, bei Anweisungstriggern nicht. Durch eine hinter `when` angegebene Bedingung kann die Abfeuerung eingeschränkt werden, was aber auch eine `if`-Bedingung am Anfang des Triggers kann.

Trigger können gelöscht werden mittels `drop trigger triggername`. Möchte man einen Trigger nur temporär außer Kraft setzen, so verwendet man `alter trigger triggername { disable | enable }`. Ein Umbenennen von Triggern ist nicht vorgesehen. Um alle Trigger für eine Tabelle zu deaktivieren oder wieder zu aktivieren, verwendet man `alter table tabellenname { disable | enable } all triggers;`.

2.1.2 Zugriff auf Daten in Triggern bei Oracle

Bei Zeilentriggern kann man auf die Pseudo-Datensätze `:old` und `:new` zugreifen, die automatisch zur Verfügung stehen. Sie sind vom Rowtyp der aktuellen Tabelle und beinhalten das aktuelle Tupel vor `:old` bzw. nach `:new` der Änderung. Bei Einfügeoperationen gibt es naturgemäß nur `:new`, bei Löschoperationen nur `:old`. Die Datensätze heißen Pseudo-Datensätze, weil man nur auf ihre Felder `:new.spielnr` zugreifen kann, aber nicht auf den Datensatz als Ganzes, um ihn z. B. einer anderen Variable vom selben Rowtyp zuzuweisen oder ihn damit zu vergleichen.

Die Werte im `:new`-Pseudodatensatz kann man verändern. Geschieht das in einem `before`-Trigger, so wird die Datenbankoperation mit den veränderten Daten weitergeführt. Das wird beispielsweise beim automatischen Hochzählen von künstlich erzeugten Schlüsseln verwendet. Ein Beispiel dazu ist im Dokument „Auto-Increment-Spalten in Datenbanken“ enthalten. In Anweisungstriggern kann man die Pseudodatensätze nicht verwenden.

Um festzustellen, von welcher Art Datenbankoperation der Trigger gerade abgefeuert wurde, kann man einige boolesche Prädikate abfragen. Um das Beispiel zur Erzeugung einer History aus dem Dokument „History-Funktion in Datenbanken“ – dort gezeigt mit Hilfe von PostgreSQL-Regeln – unter Verwendung eines Oracle-Triggers und der Prädikate zu realisieren, programmiert man so:

```
create trigger artikel_history
  before update or delete on artikel
  for each row
declare
  aktion char(1);
begin
  if updating then
    aktion := 'U';
  else
    aktion := 'D';
  end if;
  insert into artikel_history (artnr, bezeichnung, vkpreis, aktion)
  values (:old.artnr, :old.bezeichnung, :old.vkpreis, aktion);
end;
/
```

Genau wie im Beispiel aus dem History-Dokument muss die Tabelle `artikel_history` mit einer Autoincrement-Spalte versehen sein, was wiederum im entsprechenden Dokument erläutert wird und bei Oracle etwas mehr Aufwand erfordert als bei anderen Datenbanken.

2.1.3 `instead of`-Trigger

Diese Art von Triggern existieren speziell für Views, um diese auch dann schreibfähig zu machen, wenn sie Aggregatfunktionen oder Verbundoperationen enthalten. Ein Beispiel hierzu findet sich im Dokument „Views in SQL“.

2.1.4 Zugriff auf Daten der aktuellen Tabelle (mutating table)

Bei Oracle (und nach aktuellem Kenntnisstand ausschließlich dort) gibt es das sogenannte „Mutating Table“-Problem. Es ist nicht erlaubt, mittels Anweisungen innerhalb eines Zeilentriggers auf sich verändernde Tabellen schreibend oder auch nur lesend zuzugreifen. Als sich verändernde Tabellen gilt zunächst einmal die Tabelle, für die der Zeilentrigger definiert wurde, aber auch alle Tabellen, die durch ein `on delete cascade` ebenfalls verändert werden. Falls man gegen diese Regeln verstößt, bekommt man den Fehler `ORA-04091`.

> Zitat aus dem ZDNet-Artikel Learn to avoid the mutating table problem in Oracle:
> Mutating tables in Oracle can drive any IT database manager insane when it comes to tracking down the culprit. With a clear idea of the table design required, however, this problem can be avoided.

Ebenfalls verboten ist ein schreibender oder auch lesender Zugriff auf die eindeutigen Schlüssel-, Primärschlüssel- oder Fremdschlüsselattribute einer die triggernde Tabelle einschränkenden Tabelle. Dies sind die Tabellen, die zwecks Fremdschlüsselprüfung ausgelesen werden müssen.

Diese Einschränkungen gelten für alle Zeilentrigger; bei Anweisungstriggern nur dann, wenn sie auf Grund eines `delete cascade` abgefeuert wurden.

Wenn eine `insert`-Anweisung nur eine einzige Zeile betrifft, dann gilt die Tabelle nicht als sich verändernd. Falls aber `insert into` *`tabelle`* `select ...` verwendet wird, ist die *`tabelle`* eine sich verändernde Tabelle, auch wenn das Ergebnis der `select`-Anweisung nur ein einziges Tupel ist.

Das Beispiel mit der Überprüfung der Zimmerbelegung ist in PostgreSQL überhaupt kein Problem (siehe Abschnitt 2.2.1 auf Seite 28), bei Oracle erfordert es mehr Aufwand. Man muss ein Package definieren, um eine Variable zu bekommen, die länger lebt als ein Prozeduraufruf. Dann kann man in einem Anweisungstrigger nach der Durchführung aller anderen Teile der Anweisung die Konsistenz überprüfen.

Es gibt zwei Möglichkeiten, das Problem der sich verändernden Tabelle zu umgehen. Erstens kann man den Zeilentrigger zu einer autonomen Transaktion machen, zweitens kann man ein Package anlegen und sowohl Zeilen- als auch Anweisungstrigger benutzen. Wir schauen uns beide Varianten an und beurteilen danach, welche zu einer zufrieden stellenden Lösung führen.

Trigger mit autonomer Transaktion

```
create or replace trigger mut1
before insert on k_patient
for each row
declare
  anzfrei integer;
pragma autonomous_transaction;
begin
  select betten - count(*) into anzfrei
  from k_zimmer z, k_patient p
  where z.znr = p.znr
  and z.znr = :new.znr
  group by z.znr, betten;
  if anzfrei <= 0 then
    raise_application_error (-20001,
    'Zimmer ' || :new.znr || ' überbelegt');
  end if;
  commit work;
exception
  when no_data_found then
    return;
end;
/
```

Der Trigger wird hier als autonome Transaktion ausgeführt. Dabei sieht er bei der Verarbeitung jeder einzelnen Zeile den Zustand der Datenbank, wie er vor dem Beginn der

gesamten Transaktion war. Durch diesen Kunstgriff wird vermieden, dass die Tabelle `k_patient` als sich verändernde Tabelle wahrgenommen wird.

Beim Einfügen eines neuen Patienten wird also geprüft, ob noch mindestens 1 Bett im Zimmer frei ist. Die Ablauflogik unterscheidet drei Fälle: Wird beim `select` nichts gefunden, so wird die Ausnahme `no_data_found` erzeugt und anschließend gefangen. Wird festgestellt, dass zwar Betten belegt sind, es aber noch freie Betten gibt, wird die autonome Transaktion mittels `commit work` beendet, also das Einfügen durchgeführt. Im dritten Fall, dass die Variable `anzfrei` einen Wert kleiner als 0 bekommt, wird ein Anwendungsfehler erzeugt, der dazu führt, dass das Einfügen des Tupels rückgängig gemacht wird. So weit, so gut.

Die Tücke steckt aber im Detail. Fügen wir mehrere Patienten in einer einzigen Transaktion ein (evtl. sogar auf mehrere Anweisungen verteilt), so wird für jeden einzelnen Patienten geprüft, ob zu Beginn der Transaktion noch ein Bett im jeweiligen Zimmer frei war. **Das genügt aber nicht!** Denn wenn nur ein Bett frei war, kann ich dem Zimmer beliebig viele neue Patienten innerhalb der Transaktion hinzufügen – was zweifelsohne zu einer Überbelegung führen kann. Der Ansatz mit autonomer Transaktion führt also zu einer völlig inakzeptablen **Scheinlösung**!

Trigger mit Package

```
create or replace package mutpack1 as
  type zimmernr_t is table of k_zimmer.znr%type
    index by binary_integer;
  zimmernr zimmernr_t;
  anzahl   binary_integer := 0;
end;

create or replace trigger mut_row
before insert on k_patient
for each row
begin
  mutpack1.anzahl := mutpack1.anzahl + 1;
  mutpack1.zimmernr(mutpack1.anzahl) := :new.znr;
end;
/

create or replace trigger mut_stat
after insert on k_patient
declare
  anzfrei   integer;
  zimmernr  k_zimmer.znr%type;
begin
  for i in 1..mutpack1.anzahl loop
```

```
    zimmernr := mutpack1.zimmernr(i);
    select betten - count(*) into anzfrei
    from k_zimmer z, k_patient p
    where z.znr = p.znr
    and z.znr = zimmernr
    group by z.znr, betten;
    if anzfrei < 0 then
      mutpack1.anzahl := 0;
      raise_application_error (-20001,
      'Zimmer ' || zimmernr || ' überbelegt');
    end if;
  end loop;
  mutpack1.anzahl := 0;
end;
/
```

Der zweite Ansatz ist leider etwas aufwendiger. Es wird ein Package generiert, das eine tabellenwertige Variable enthält und eine Variable mit der Anzahl der Tupel in dieser Variablen.

Jede Transaktion, die das Package verwendet, sieht eine eigene Instanz dieser Variablen, so dass parallele Transaktionen hier kein Problem darstellen.

Es werden zwei Trigger verwendet: ein Zeilen- und ein Anweisungstrigger. Der Zeilentrigger fügt die Nummer des betroffenen Zimmers zum Array `zimmernr` hinzu und erhöht die Variable `anzahl` um 1.

Der Anweisungstrigger, der ausgeführt wird, nachdem alle Tupel aus der Anweisung eingefügt worden sind, hat kein Problem mit der sich verändernden Tabelle. Er durchläuft das Array und prüft für jede dort gespeicherte Zimmernummer (evtl. also mehrfach, falls mehrere Patienten in dasselbe Zimmer gekommen sind, aber das stört nicht weiter), ob mittlerweile die Anzahl der freien Betten unter 0 gesunken ist. Sollte das der Fall sein, wird ein Anwendungsfehler generiert und damit die gesamte Anweisung zurückgefahren.

Für den Fall, dass es gut gegangen ist, wird die Package-Variable `anzahl` wieder auf 0 gesetzt für einen evtl. zweiten Durchlauf innerhalb derselben Transaktion. Bei einem Ende der Transaktion werden die Variablen des Package ohnehin gelöscht.

Nur diese Variante liefert also eine sinnvolle Überprüfung – für den Fall des *Einfügens* neuer Patienten. (Die Konsistenz der Datenbank diesbezüglich ist mit diesem einen Package mit seinen zwei Triggern aber noch nicht gesichert, denn auch beim Verlegen von Patienten von einem Zimmer in ein anderes kann es zu Überbelegungen kommen.)

2.2 Trigger bei PostgreSQL

Nicht für alle automatisch ausgelösten Aktionen ist es bei PostgreSQL notwendig, einen Trigger zu verwenden. Um beispielsweise Views schreibfähig zu machen oder eine History-

Funktion einzuführen, genügen hier die sogenannten „Rules“, siehe auch die Dokumente „Views in SQL“ und „History-Funktion in Datenbanken“. Rules ersetzen oft die `instead of`-Trigger von Oracle.

Trigger werden in PostgreSQL mit der Anweisung `create trigger` erzeugt. Im Gegensatz zu Oracle kann hier aber kein Prozedurrumpf direkt mit angegeben werden. Statt dessen steht hier immer ein `execute procedure`, das eine Funktion aufruft, die bereits definiert sein muss. Ein kleines Beispiel für eine Triggerfunktion findet sich in Abschnitt 1.2.8 auf Seite 20.

2.2.1 Erzeugen von Triggern in PostgreSQL

Die Syntax für die Erzeugung eines Triggers lautet:

```
create trigger triggername
  { before | after }
  { insert | update | delete [ or { insert | update | delete } ... ] }
  on tabellenname
  for each { row | statement }
  execute procedure funktionsname ( [ parameter1 [, parameter2, ... ] ] )
```

Viele Trigger übergeben keine Parameter an die Funktion, aber die leeren Klammen müssen trotzdem angegeben werden. Als Parameter sind beliebige Zeichenkettenliterale möglich. Ein einfacher Trigger, passend zur Funktion `freibetten()` auf Seite 20, wäre:

```
create trigger insert_patient
  after insert on patient
  for each row execute procedure freibetten();
```

Bei einem Einfügen eines Patienten wird durch die Funktion geprüft, dass das Zimmer mit diesem Patienten nicht überbelegt wird. (Die Konsistenz der Datenbank diesbezüglich ist mit diesem einen Trigger aber noch nicht gesichert.)

Bezogen auf ein Ereignis können auch mehrere Trigger definiert werden. Laut SQL-Standard werden die Trigger in der Reihenfolge ihrer Erzeugung abgefeuert – bei PostgreSQL geschieht es aber in alphabetischer Reihenfolge nach dem Triggernamen, weil das leichter nachzuvollziehen ist. Wenn ein Zeilen-`before`-Trigger das `new`-Tupel verändert, sehen die nachfolgend abgefeuerten Trigger (und auch die abfeuernde SQL-Anweisung) nur dieses veränderte Tupel.

Dadurch, dass Triggerfunktionen weitere SQL-Anweisungen beinhalten können, können weitere Trigger ausgelöst werden, was zu einem regelrechten Dominoeffekt führen kann. Es ist die Verantwortung des Programmierers, hier keine Endlosschleifen einzubauen.

Trigger können mit `drop trigger triggername on tabellenname;` gelöscht werden. Da Triggernamen im Standard nicht nur bezüglich eine Tabelle eindeutig sein müssen, ist bei anderen Datenbanken (und im Standard) die Angabe der zugehörigen Tabelle nicht notwendig.

Deaktivieren kann man Trigger nicht, aber da sie keinen Code enthalten (der steckt ja in der Funktion) und der Erstellungszeitpunkt ohne Bedeutung ist, kann man sie leicht löschen und neu erzeugen. Umbenennen kann man Trigger mittels `alter trigger triggername on tabellenname rename TO neuername`, was für das Festlegen der Abfeuerungsreihenfolge evtl. notwendig ist.

2.2.2 Besonderheiten der Triggerfunktionen in PostgreSQL

Einige spezielle Variablen werden automatisch bei Triggerfunktionen erzeugt:

- `new` vom Typ `record` (Datensatz, Zeile). Hierin ist das neue Tupel bei `insert`- und `update`-Zeilentriggern enthalten.
- `old` vom Typ `record` (Datensatz, Zeile). Hierin ist das alte Tupel bei `update`- und `delete`-Zeilentriggern enthalten.
- `TG_NAME` vom Datentyp `name` (eine Zeichenkette). Hierin ist der Name des ausgelösten Triggers enthalten, denn dieselbe Funktion könnte ja von verschiedenen Triggern aufgerufen werden.
- `TG_WHEN` vom Datentyp `text`. Der Inhalt lautet `BEFORE` oder `AFTER`, je nach Abfeuerungszeitpunkt des Triggers.
- `TG_LEVEL` vom Datentyp `text`. Der Inhalt lautet `ROW` oder `STATEMENT`, je nach Definition des Triggers.
- `TG_OP` vom Datentyp `text`. Der Inhalt lautet `INSERT`, `UPDATE` oder `DELETE` – je nach ausgeführter Operation.
- `TG_RELID` vom Datentyp `oid` (object id). Der Inhalt ist die interne Nummer der Tabelle, die den Trigger abgefeuert hat.
- `TG_RELNAME` vom Datentyp `name` (eine Zeichenkette). Der Inhalt ist der Name der Tabelle, die den Trigger abgefeuert hat.
- `TG_NARGS` vom Datentyp `integer`. Der Inhalt ist die Anzahl der Parameter, die der Funktion bei der Definition des Triggers mitgegeben wurde (auch wenn die Funktion selbst als parameterlos deklariert werden muss).
- `TG_ARGV` vom Datentyp `array of text`. Dies sind die Parameter aus der `create trigger`-Anweisung. Die Elemente werden ab 0 gezählt.

Eine Triggerfunktion gibt immer `null` zurück oder ein Tupel vom Typ der Tabelle, die sie über den Trigger aufgerufen hat – also meistens `return new;`. Gibt ein Zeilen-`before`-Trigger `null` zurück, so werden keine weiteren Trigger aufgerufen und die zugehörige Datenbankoperation für dieses Tupel auch nicht durchgeführt. Wird ein Tupel zurückgeliefert,

so wird die Operation mit diesem Tupel weitergeführt – evtl. also nicht mehr mit den Werten, die bei der `insert`- oder `update`-Operation ursprünglich angegeben wurden.

Der Rückgabewert der Funktion bei allen Statement- und `after`-Triggern ist für die Ausführung der eigentlichen Datenbankoperation ohne Bedeutung; bei Rückgabe von `null` werden aber die folgenden Trigger nicht mehr abgefeuert – schließlich kann es mehrere Trigger pro Ereignis geben. Bei allen Arten von Triggern führt das Erzeugen einer Ausnahme mittels `raise exception` zum Abbruch der Gesamtoperation und damit der Transaktion.

Triggerfunktionen können übrigens auch in C oder anderen Sprachen geschrieben werden (aber nicht in Java[1]) – hierauf wird in diesem Dokument aber nicht eingegangen.

2.2.3 Zugriff auf Daten der aktuellen Tabelle

Das berühmt-berüchtigte „Mutating Table“-Problem existiert bei PostgreSQL nicht. Die zugehörige Frage lautet nicht: „Warum gibt es das Problem bei PostgreSQL nicht?“, sondern „Warum gibt es das Problem überhaupt, und ausschließlich bei Oracle?“

Die Regeln für die Sichtbarkeit von Änderungen sind einfach: Alle Änderungen, die zum jeweiligen Zeitpunkt an irgendwelchen Daten von der eigenen Transaktion vorgenommen worden sind, sind sichtbar und alle Operationen wirken auf den aktuellen Zustand. So kann man durchaus in einer `after row`-Triggerfunktion bei einer `insert`-Anweisung das soeben eingefügte Tupel mittels `delete` wieder löschen oder mittels `update` ändern, oder auch mittels irgendwelcher `select`-Anweisungen in einem `after`-Trigger Konsistenzprüfungen bezüglich der veränderten Tabelle vornehmen – wie das im Beispiel mit der Zimmerbelegung im Krankenhaus auch passiert.

Vorsicht ist evtl. geboten bei `insert`-Triggern und `select ... into ...`-Anweisungen, weil man die Reihenfolge der Ergebnistupel des `select` nicht unbedingt kennt. Aber beim `after statement`-Trigger sind auf jeden Fall alle Daten verarbeitet..

$Id: trigger.tex,v 68590d482bd2 2009/11/13 15:30:57 bibjah $

1) Das könnte mit folgendem zusammenhängen: `http://programming.newsforge.com/programming/04/04/07/2021242.shtml`