HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Datenbanksysteme 2

Kap. 7: DB-interne Programmierung

Prof. Dr. Carsten Kleiner

Wo sind wir?

Fortgeschr. SQL

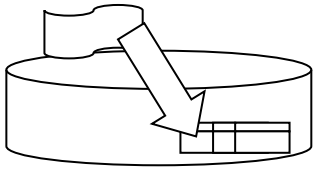
```
SELECT row_number()
    OVER (PARTITION BY year
          ORDER BY rating DESC),
    mr.*
FROM mr
```

TX (Anw)

```
BEGIN TX
IF success
  COMMIT;
ELSE
  ROLLBACK;
```

Relationaler DB-Zugriff

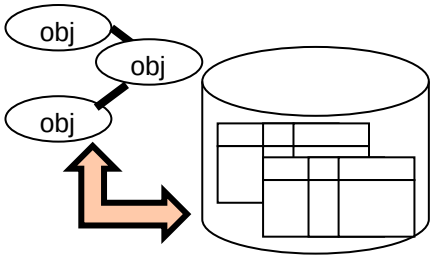
JDBC + TXn, Embedded SQL



Transaktionen (intern)

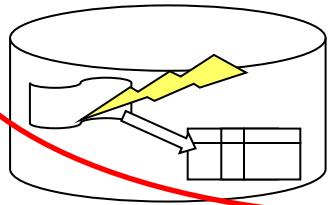
```
T1:                T2:
Read(X);           Read(X);
X := X-10;         X := X+15;
Write(X);          Write(X);
Read(Y);
 . . .
```

OR Mapper

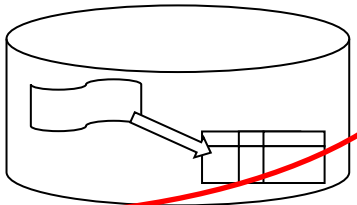


DB-interne Programmierung

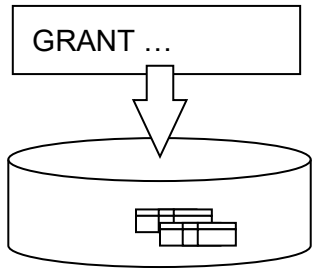
Stored Procedures



Aktive DBMS/Trigger



DB-Sicherheit



Inhalt

Stored Procedures mit PL/SQL bzw. PL/pgSQL

Stored Procedure mit Java

Aktive Datenbanksysteme und Trigger

Stored Procedures

Statisches SQL:

- SQL in bekannte Sprache (z.B. C oder Java) einbetten

Andere Herangehensweise:

- SQL mit prozeduralen Elementen erweitern
- D.h. neue Sprache, die SQL-zentriert ist, entwerfen

Beispiel für die Vorlesung:

- PL/SQL (SQL-Erweiterung von Oracle) bzw. PL/pgSQL (SQL-Erweiterung von PostgreSQL)
- Leider nicht normiert, jede DB unterstützt eine andere Sprache

Wichtige Kennzeichen:

- Sprache wird von der Datenbank direkt ausgeführt
- Programmcode wird in der Datenbank gespeichert
- Daher "Stored Procedures": "Gespeicherte Prozeduren"

Stored Procedures: Einleitung

Was sind gespeicherte Prozeduren bzw. „Stored Procedures“?

- **Stored Procedures** sind alleinstehende Einheiten, die (in einer Datenbank) global definiert und verfügbar sind
- **Stored Procedures** lassen sich in proprietären Datenbanksprachen (z. B. PL/SQL oder PL/pgSQL) definieren und in der Datenbank ablegen
- Als Stored Procedures werden in der Regel **Prozeduren** (ohne Rückgabetyp) **und Funktionen** (mit Rückgabetyp) bezeichnet
- Als Abkürzung für Stored Procedures wird **SP** verwendet

Beispiel: Arbeitszeitenverbuchung (Oracle)

```
create or replace
procedure day_finished(p_employee_id number, p_hours number) as
    v_old_hours number;
    v_hours_per_day number;
    v_hours number;
begin
    select work_hours into v_old_hours from work_hour
    where employee_id = p_employee_id;

    select hours_per_day into v_hours_per_day from employee
    where employee_id = p_employee_id;

    v_hours := v_old_hours + p_hours - v_hours_per_day;

    update work_hour set work_hours = v_hours
    where employee_id = p_employee_id;
end;
```

Vergleich Beispiel für PostgreSQL

```
CREATE OR REPLACE
PROCEDURE day_finished(p_employee_id NUMERIC, p_hours NUMERIC) AS $$
DECLARE
    v_old_hours NUMERIC;
    v_hours_per_day NUMERIC;
    v_hours NUMERIC;
BEGIN
    SELECT work_hours INTO v_old_hours FROM work_hour
    WHERE employee_id = p_employee_id;

    SELECT hours_per_day INTO v_hours_per_day FROM employee
    WHERE employee_id = p_employee_id;

    v_hours := v_old_hours + p_hours - v_hours_per_day;

    UPDATE work_hour SET work_hours = v_hours
    WHERE employee_id = p_employee_id;
END;
$$ LANGUAGE plpgsql;
```

Demo

- Kompilieren und Aufruf der Stored Procedure

- Aufruf aus IDE oder DB-Tool

```
EXECUTE day_finished(1, 9);
```

Bzw.

```
CALL day_finished(1, 9);
```

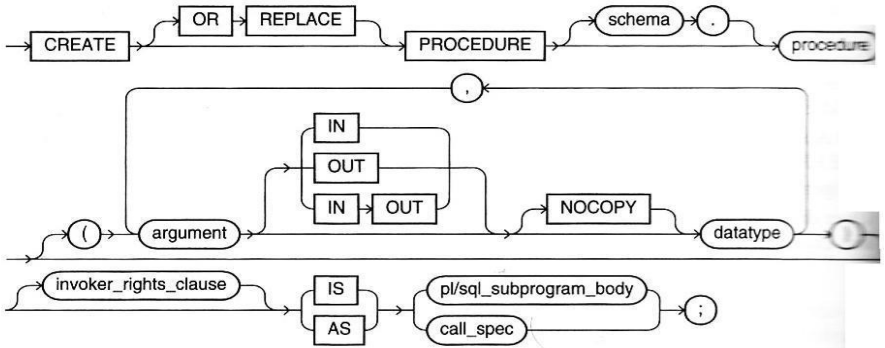
- Entwicklung in IDE oder DB-Tool

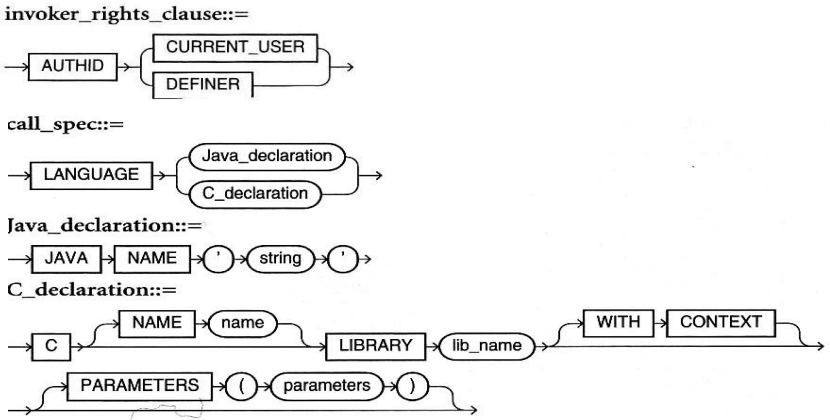
Stored Procedures Oracle – Syntax Prozedur

CREATE PROCEDURE

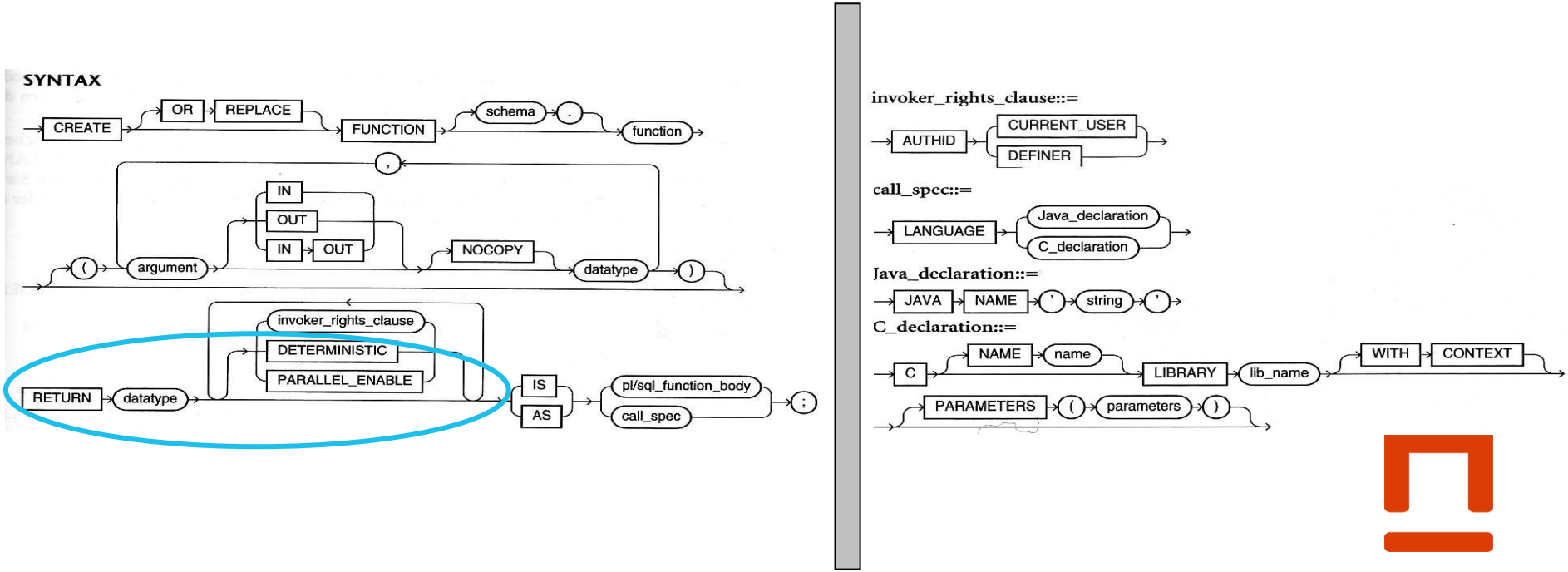
SIEHE AUCH ALTER PROCEDURE, BLOCKSTRUKTUR, CREATE LIBRARY, CREATE FUNCTION, CREATE PACKAGE, DATENTYPEN, DROP PROCEDURE, Kapitel 25 und 27

SYNTAX





Stored Procedures Oracle – Syntax Funktion



Stored Procedures PL/pgSQL

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

Eigentlicher Code bei PL/pgSQL ist dann in dem String `definition` :

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Stored Functions PL/pgSQL

Analog zu Procedures, nur mit Rückgabewert/-klausel

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

PL/SQL bzw. PL/pgSQL: Aufbau

- grundlegende Einheit von PL/SQL bzw. PL/pgSQL ist ein **Block**
- ein Block besteht aus
  - Deklarationsabschnitt (DECLARE),
  - Anweisungsteil (BEGIN)
  - und einem Fehlerbehandlungsabschnitt (EXCEPTION)
- Blöcke können ineinander geschachtelt sein
- Nur Oracle: bei Stored Procedures werden die Variablen nach dem Schlüsselwort IS bzw. AS definiert

PL/SQL bzw. PL/pgSQL: Variablen und Datentypen

Variablen werden im DECLARE-Teil definiert

- Syntax: variable_name type [CONSTANT] [NOT NULL] [:=value];

vorhandene Datentypen sind

- Skalare Typen, z. B. DATE, NUMERIC
- Zusammengesetzte Typen (RECORD, [TABLE, VARRAY])
- Verweise auf Cursor (REF CURSOR)
- ...

Mit %TYPE kann der Datentyp einer Datenbankspalte referenziert werden

- z.B. students.first_name%TYPE

PL/SQL bzw. PL/pgSQL: Kontrollstrukturen

IF-THEN-ELSE-END IF

Schleifen

- Einfache Schleifen
  - LOOP sequence_of_statements END LOOP;
  - Abbruchbedingung: EXIT [WHEN condition];
- WHILE-Schleifen
- Numerische FOR-Schleifen

Nur Oracle**: GOTOs** und Marken

- Syntax eines GOTOs: GOTO label;
- Syntax Markendefinition: <<markenname>>

PL/SQL bzw. PL/pgSQL: Anlegen

- Definition der SP muss in die Datenbank eingepflegt werden (z.B. per SQL Developer, pgAdmin oder IDE)

- DBS parst die SP und speichert die „geparste“ Repräsentation in der DB

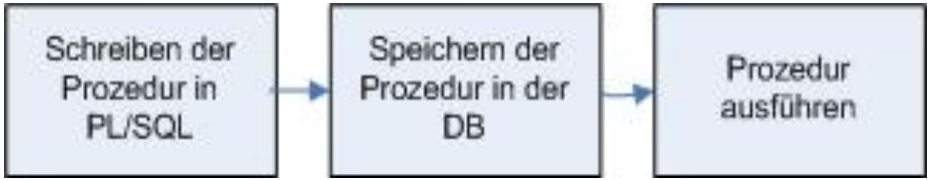
- Veränderung über passenden Editor in SQL Developer bzw. pgAdmin oder per erneutem Einpflegen wie oben ("`CREATE OR REPLACE ...`")

PL/SQL bzw. PL/pgSQL: Ausführung

Stored Procedures lassen sich in verschiedenen Formen ausführen:

- **interaktiv** mit SQL Developer, pgAdmin oder IDE
  - EXECUTE prozedurname (parameterwerte);
  - CALL prozedurname(parameterwerte);
- **innerhalb von PL/SQL bzw. PL/pgSQL-Blöcken** wie normale Prozeduren aufrufbar
- in **SQL-Anweisungen** (nur Funktionen)
- In **Pro*C/ECPG**
  - über EXEC SQL EXECUTE mit aktuellen Parametern versehen und starten (wie normale Prozeduren)
- Über **JDBC** aus Java heraus
- ...

PL/SQL bzw. PL/pgSQL: Ablauf



1. Schritt: **Schreiben** der Prozedur bzw. der Funktion in PL/SQL bzw. PL/pgSQL
   - Kann z.B. in einer Skript-Datei (*.sql) geschehen

2. Schritt: **Speichern** der Prozedur bzw. der Funktion in der Datenbank
   - Durch Ausführen der Skript-Datei in SQL Developer, pgAdmin
   - alternativ über eine Entwicklungsumgebung

3. Schritt: **Ausführen**
   - Prozedur: als eigenständiger Aufruf
   - Funktion: als Teil eines Ausdrucks

Beispiel einer "Stored Function"

```
create or replace
  function get_extra_hours
    (p_employee_id in employee.employee_id%type)
    return number is
  v_hours number;
begin
  select work_hours into v_hours
  from work_hour
  where employee_id = p_employee_id;

  return v_hours;
end;
```

Vergleich Beispiel PL/pgSQL

```
create or replace
  function dbs2.get_extra_hours
    (p_employee_id in dbs2.employee.employee_id%type)
  RETURNS numeric as $$
    DECLARE
    v_hours numeric;
begin
  select work_hours into v_hours
  from dbs2.work_hour
  where employee_id = p_employee_id;
  return v_hours;
end;
$$ LANGUAGE plpgsql;
```

Aufruf einer "Stored Function"

Über SELECT:

```
select get_extra_hours(1) [from dual];
```

In beliebigem anderem SQL-Select:

```
select e.*, get_extra_hours(e.employee_id)
from employee e;
```

Aus PL/SQL bzw PL/pgSQL heraus ("anonymer Block"):

```
dbms_output.put_line('Std: '||get_extra_hours(1));
```

BZW.

```
RAISE NOTICE 'Std: %', get_extra_hours(1);
```

Stored Procedures - Aspekte

- Möglichkeit, **Logik** in die Datenbank zu legen
- **Modularität**
- **Wiederverwendung**
  - effizientere Entwicklung
  - Kostenersparnis
- **zentrale Wartung**
  - alle gespeicherten Prozeduren liegen in der Datenbank
- Für bestimmte Aufgaben sehr gute **Performance**
- **Proprietäre Sprachen**
  - sind teils nur über proprietäre Sprachen zu definieren

Gespeicherte Prozeduren - Fragenblock

Diskutieren Sie 3 Minuten die folgenden Fragen mit Ihrem Nachbarn / Ihrer Nachbarin:

Worin liegen die Vorteile/Nachteile von Stored Procedures?

Überlegen Sie anhand der folgenden Stichpunkte:

- Information Hiding
- Änderungen am Datenmodell
- Fehlerbehandlung, Konsistenz
- Rechenzeit / Speicherbedarf auf Clients
- Sicherheit
- Benötigtes Know-How bei EntwicklerInnen
- Schichtentrennung
- Wartung der Anwendung

Zusammenfassung Erster Teil

- “Stored Procedures“ sind Codeblöcke, die in der Datenbank gespeichert werden und von der Datenbank ausgeführt werden
- Als Sprache wird eine meist vom DBMS abhängige, also proprietäre, Sprache verwendet
- Unsere Beispiele: PL/SQL von Oracle, pg/plsql von PostgreSQL
- Die Sprachen wurde als prozedurale Erweiterung von SQL erstellt
- Daher ist die Integration von SQL in die Sprache sehr eng
  - Z.B. sind die PL/SQL-Datentypen die Oracle-Datentypen bzw. die pg/plsql-Datentypen i.w. die PostgreSQL-Datentypen, inkl. `NULL`

Inhalt

Stored Procedures mit PL/SQL bzw. PL/pgSQL

Stored Procedures mit Java

Aktive Datenbanksysteme und Trigger

Stored Procedures mit Java

- Müssen SPs unbedingt mit einer proprietären Sprache geschrieben werden?
- Nein, viele Datenbanken bieten auch die Möglichkeit, Java als SP-Sprache zu verwenden (hier: Oracle Ja, PostgreSQL Nein [dafür aber Python])
- Problem dabei: Integration mit SQL ist weniger gut
- Vergleichbar mit normalen Java-Code (JDBC), der aber direkt von der Datenbank ausgeführt wird.

Java SPs: Sinnvolle Verwendungen

Daten aus der Datenbank werden algorithmisch aufwendig verarbeitet

Beispiel 1: Rechnungen / Buchungen erstellen aus Warenkörben von Internet-Shops

- Selektieren der Daten aus den Warenkorb-bezogenen Tabellen
- Sammeln der Daten für Rechnungserstellung
  (Betrag, Kunde, Adresse, Zahlverfahren)
- Insert der entsprechenden Daten in Rechnungstabelle
- Verbuchung von Lastschriften in Finanzbuchhaltung über Fibu-API

Beispiel 2: Erstellen von Statistiken

- Sammeln aller erforderlichen Daten in den Tabellen
- Durchführung komplexer Berechnungen

Java Stored Procedures - Szenarien

Szenario A. Java-Programme auf Client verarbeiten SQL-Befehle

- Fat Client führt gesamte Aktion durch: iteriert komplett durch `ResultSet`
- Nachteile
  - hoher Transfer zwischen Server und Client (Treffermengen), damit schlechte Performance / Skalierbarkeit
  - oft werden Möglichkeiten von SQL nicht genutzt!

Szenario B. Erweiterung von SQL um programmiersprachliche Elemente

- Stored Procedures in DB gespeichert und von Client aufgerufen
- Nachteil: proprietäre Sprachen (in Oracle: PL/SQL)

Java Stored Procedures - Szenarien

Szenario C. Java-Programme in DB führen SQL-Befehle aus

- „normale“ Java-Klassen auf DB-Server: Methoden beinhalten SQL
- Clients greifen auf SP-Methoden zu über entfernte Aufrufe
- Java-Klassen sind in Datenbank gespeichert, Aufruf vergleichbar mit PL/SQL-SPs
- Bewertung später

Java Stored Procedures - Ablauf

Schritt: Schreiben der Server-Klasse in Java

- besitzt statische Methoden, die SQL-Befehle (z.B. JDBC) enthalten
- wird normal kompiliert

Schritt: Java-Klasse (Source oder Bytecode) in DB laden

- mit Lade-Tool der Datenbank oder über DB-Befehl

Schritt: Stored Procedure als Wrapper erzeugen

- per SQL-Befehl

Schritt: Schreiben eines Clients

- ruft Stored Procedure (z.B. per JDBC) auf

Oracle besitzt integrierte Java VM

analog für andere DBMS (z.B. DB2) verfügbar, aktuell jedoch nicht in PostgreSQL

Java Stored Procedures - Beispiel

Schritt 1: Schreiben der Server-Klasse in Java und Laden in die Datenbank

```
CREATE OR REPLACE JAVA SOURCE NAMED "Employee" AS
// importe ...
public class Employee {
    public static void dayFinished(long employee_id, long hours) throws SQLException {
        Connection connection = new OracleDriver().defaultConnection();
        long old_hours, hours_per_day;
        String sql = "SELECT work_hours FROM work_hour WHERE employee_id = ?";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
             stmt.setLong(1, employee_id);
             try (ResultSet rs = stmt.executeQuery()) {
                 rs.next();
                 old_hours = rs.getLong("work_hours");
             }
        }
        sql = "SELECT hours_per_day FROM employee WHERE employee_id = ?";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
             stmt.setLong(1, employee_id);
             try (ResultSet rs = stmt.executeQuery()) {
                 rs.next();
                 hours_per_day = rs.getLong("hours_per_day");
             }
        }
        sql = "UPDATE work_hour SET work_hours = ? WHERE employee_id = ?";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
             stmt.setLong(1, old_hours + hours - hours_per_day);
             stmt.setLong(2, employee_id);
             stmt.executeUpdate();
        }
    }
}
```

Java Stored Procedure – Beispiel

Schritt 2: Java-Klasse in Datenbank kompilieren

```
alter java source "Employee" compile
```

Schritt 3: Stored Procedure als Wrapper erzeugen

```
CREATE OR REPLACE
PROCEDURE day_finished_java (employee_id IN number,
                             hours IN  number)
AS LANGUAGE JAVA NAME
          'Employees.dayFinished(long, long)';
```

Schritt 4: Aufrufen der Prozedur (wie PL/SQL), Bsp:

```
execute day_finished_java(1, 9);
```

Java Stored Procedures - Bewertung

Vorteile

- Alle Vorteile von SPs generell
  - Z.B. Performance etc., da in der DB ausgeführt
- Java wird als einzige und moderne Sprache verwendet
  - Flexibilität: Transformation zu anderen Architekturen möglich
  - Komplexere Berechnungen oder Strukturen besser nutzbar als in PL/SQL

Nachteile

- Verwendung etwas umständlich (Wrapper)
- Keine „echte“ Objektorientierung
- Immer noch herstellerspezifische, proprietäre Schnittstelle, aber für verschiedene DBMS verfügbar und „recht ähnlich“

Zusammenfassung

Stored Procedures (Prozeduren und Funktionen)

- Syntax
- Ablauf
- Bewertung

Java Stored Procedures

- Java-Programme durch Stored Procedures gekapselt
- Ablauf
- Bewertung

Referenzen: Stored Procedures

- Jürgen Sieben: "Oracle PL/SQL: Das umfassende Handbuch für Datenbankentwickler. Aktuell zu Oracle 19c und 21c “, Rheinwerk Computing; 2023
- Oracle: "Java Stored Procedures Application Example", https://docs.oracle.com/en/database/oracle/oracle-database/21/jjdev/Java-stored-procedure-application-example.html (15.11.2024)
- Oracle Handbücher online: https://docs.oracle.com/en/database/oracle/oracle-database/21/books.html (15.11.2024)

Inhalt

Stored Procedures mit PL/SQL bzw. PL/pgSQL

Stored Procedure mit Java

Aktive Datenbanksysteme und Trigger

Aktive Datenbanksysteme

Szenarien

Kontrolle des Lagerbestands

- Nachbestellen, wenn Mindestmenge erreicht

Warteliste für Reisen

- Buchung auslösen, wenn Reise verfügbar
- Verschiedene Ursachen: Absagen oder neue Angebote

Lösungsansätze?

Aktive Datenbanksysteme

Weitere Lösung: Verwendung eines aktiven Datenbanksystems!

Was ist ein aktives Datenbanksystem?

> Ein Datenbanksystem ist *aktiv,* wenn es auf (externe oder interne) *Ereignisse* durch (externe oder interne) *Aktionen* reagiert.

Das Datenbanksystem speichert **Regeln**, die definieren, **welche** Aktionen **wann** durchgeführt werden sollen.

Aktive Datenbanksysteme

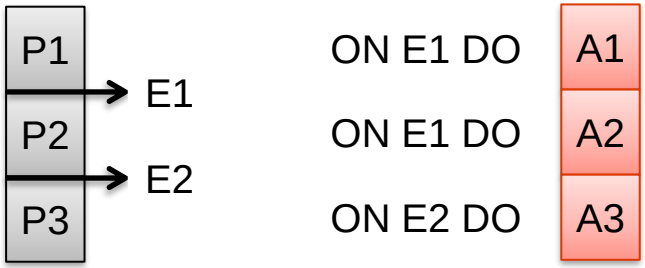
- Das Verhalten eines aktiven Datenbanksystems wird beschrieben durch aktive Regeln (Trigger) der Form
  - ON Ereignis DO Aktion
- Ein Ereignis ist etwas, das zu einem Zeitpunkt stattfindet.
  Beispiele:
  - Beginn oder Abschluss einer (Datenbank-)Operation
  - Montag, 05.10.2009, 10:35
  - 2 Stunden nach Eingang einer Bestellung
- Eine Aktion hat eine interne oder externe Wirkung.
  Beispiele:
  - Datenbankoperation(en)
  - Beginn der Auslieferung

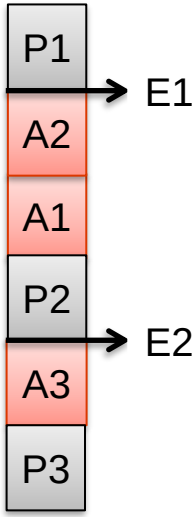
Aktive Datenbanksysteme

- Ein aktives Datenbanksystem beinhaltet (implizit oder explizit) Dienste zum
  - Anlegen, Aktivieren, Deaktivieren, Löschen von Ereignissen, Aktionen und Regeln
  - Überwachen von Ereignissen
    - *Auswahl und Auslösen von Aktionen*
- Regeln werden häufig auch in folgender Form spezifiziert:
  - ON Ereignis IF Bedingung DO Aktion
- Dies entspricht:
  - ON Ereignis DO (IF Bedingung THEN Aktion)

Aktive Datenbanksysteme

Modifikation des Programmablaufs:





Durch Eintreten von Ereignissen während des Programmablaufs wird dieser dynamisch modifiziert

Die Reihenfolge der ausgelösten Aktionen ist evtl. nicht deterministisch

Vorteile einer aktiven Datenbank

Anwendungsprogrammierung wird entlastet

- Weniger Wissen im Anwendungscode, mehr Wissen in der Datenbank; dadurch daten-nahe Logik nur einmal zu implementieren
- Erhöhte Flexibilität, z.B. im Fehlerfall
  - Modifikation der Ablaufbeschreibung
  - Modifikation der Regelbasis

Flexibilität

- Flexible Verknüpfung von Prozessschritten über aktive Regeln

Diskussion: Nachteile einer aktiven Datenbank

Diskutieren Sie mit Ihren Nachbarn, welche Nachteile einer aktiven DB Sie sich vorstellen können.

Einsatzmöglichkeiten

Aktive Regeln können eingesetzt werden für

- **Steuerung von Abläufen (Workflows)**
  - Benachrichtigung bei Fehler, Terminüberschreitung
  - Überwachung von Ausnahmesituationen
  - Aktualisierung redundanter / abgeleiteter Daten bei Änderungen
- **Vorteile:**
  - Entlastung der Anwendungsprogramme
  - Vermeiden von Polling auf der Datenbank
  - Flexibler Kontrollfluss

Stand der Technik

Kommerzielle relationale Datenbanksysteme (DB/2, PostgreSQL, Oracle, ...) und einige andere Arten von Datenbanksystemen bieten einfache Realisierungen aktiver Regeln

- Manchmal Effizienzprobleme oder Entwurfsprobleme
- Trotzdem: Konkrete anwendungsorientierte Vorschläge zum Einsatz aktiver Datenbanksysteme
- In der Praxis bewährt: einfache „Trigger“ bes. „SQL-Trigger“

Anmerkungen:

- Forschungs-Prototypen (Schwerpunkt ‘90er) boten/bieten weitergehende Möglichkeiten
- (Aktive) Regelelemente werden teils auch in verteilten (Objekt)systemen (z.B. CORBA, BPEL) angeboten

SQL-Trigger

- Zur Verknüpfung von benutzerdefinierten Aktionen mit den Standard-DB-Operationen
- Formulierung durch (einfache)
  "Event-Condition-Action Rules" (ECA-Regeln)
  - Wenn ein Ereignis eintritt, überprüfe ob die Bedingung erfüllt ist. Falls ja, dann führe die zugehörige Aktion aus.
- Die Ausführungsreihenfolge bei mehreren ausgelösten Triggern ist im Allgemeinen *nicht-deterministisch*.
- Trigger werden von verschiedenen Herstellern angeboten, jedoch noch oft mit deutlich unterschiedlicher Syntax
  (aber: in SQL:2003 sind Trigger spezifiziert)

ECA – Trigger: Beispiel

```
before update
  of work_hours
  on work_hour
for each row
```

← Event

```
when (new.work_hours > 30)
```

← Condition

Action ↓

```
begin

  insert into extra_salary values (
    :new.employee_id, (:new.work_hours-30)*50
  );

  :new.work_hours := 30;
end;
```

ECA – Trigger: Demo

Demonstration eines Triggers

- **Event**: Trigger wird immer dann ausgelöst, wenn in der Tabelle `work_hour` die Spalte `work_hours` aktualisiert wird.

- **Condition**: Genau dann, wenn die aktuellen Überstunden 30 überschreiten, wird die Aktion ausgeführt.

- **Action**: Es wird eine Auszahlung der Überstunden > 30 veranlasst und die Anzahl Überstunden auf 30 zurückgesetzt.

Demo

Demo zum Trigger Beispiel Überstunden

Demo zum Trigger Beispiel Änderungen automatisiert protokollieren

Trigger - Ereignistypen

- **Zeitereignisse**
  - absolut, relativ, periodisch
- **Datenbankereignisse**
  - Beginn oder Ende von INSERT, UPDATE, DELETE
- **DBMS Ereignisse**
  - DDL Kommandos: z.B. ALTER, DROP, CREATE, ...
  - Systemereignisse: z.B. BEFORE SHUTDOWN, AFTER LOGIN, ...

Trigger-Syntax in Oracle

```
<trigger> ::=
    CREATE TRIGGER <triggername>
    (BEFORE | AFTER) <events>
    WHEN (<condition>)
       <pl/sql-block>;          // Im wesentlichen beliebige PL/SQL-Anweisungen
                                // if, while, Ausgaben, . . .
                                // Bem.: nicht “Standard compliant”, aber
                                //       i.w. “functional equivalent”

<events> ::=
    <dml-event> {OR <dml-event>} ON <table> [FOR EACH ROW]

<dml-event> ::=
    INSERT | DELETE | UPDATE OF <column> {, <column>}

Bem.: Im SQL-Developer Trigger-Definitionen mit “/” abschließen
```

Trigger-Syntax in Oracle

```
<trigger> ::=
    CREATE TRIGGER <triggername>
    (BEFORE | AFTER) <events>
    WHEN (<condition>)
       <pl/sql-block>;          // Im wesentlichen beliebige PL/SQL-Anweisungen
                                // if, while, Ausgaben, . . .
                                // Bem.: nicht "Standard compliant", aber
                                //       i.w. "functional equivalent"

<events> ::=
    <dml-event> {OR <dml-event>} ON <table> [FOR EACH ROW]

<dml-event> ::=
    INSERT | DELETE | UPDATE OF <column> {, <column>}

Bem.: Im SQL-Developer Trigger-Definitionen mit "/" abschließen
```

Trigger-Syntax in PostgreSQL

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } {
event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Trigger-Syntax

Wesentliche Unterschiede zwischen Oracle und PostgreSQL

- PostgreSQL erfordert externe Definition einer aufzurufenden Trigger-Funktion
  - ohne Parameter [bekommt automatisch zahlreiche Umgebungsvariablen wie NEW]
  - RETURNS trigger
  - Dadurch Trennung Event und Condition von Action
  - Nutzung derselben Action in mehreren Triggern möglich
- PostgreSQL kann zusätzlich auf new/old Abbild einer gesamten Tabelle nach/vor der Änderung zugreifen: hilfreich für Konsistenz-Prüfungen auf Tabellenebene
- PostgreSQL hat spezielle CONSTRAINT Trigger zur Umsetzung komplexer Constraints, die am Ende des Statements oder der Transaktion gefeuert werden können (Funktion ähnlich mit Oracle umsetzbar, aber implizit)

Zeilen-Trigger und Statement-Trigger

Update-Statement:

```
UPDATE PERSONEN
SET Matrikel = 'I-' || Matrikel
WHERE Abteilung = 'Informatik';
```

ID	Name	Abteilung	Matrikel
1	Meier	Informatik	1234
2	Müller	Wirtschaft	5678
3	Bauer	Informatik	1133
4	Schulz	Wirtschaft	2244

Zeilen-Trigger und Statement-Trigger

Zeilenbasierter Trigger ("FOR EACH ROW")

```
create or replace
TRIGGER trg_personen
AFTER UPDATE ON personen
FOR EACH ROW
BEGIN
    -- Aktionen, kann z.B. :new.name verwenden
END;
```

ID	Name	Abteilung	Matrikel
1	Meier	Informatik	1234
2	Müller	Wirtschaft	5678
3	Bauer	Informatik	1133
4	Schulz	Wirtschaft	2244

Zeilen-Trigger und Statement-Trigger

Statement Trigger (ohne "FOR EACH ROW")

```
create or replace
TRIGGER trg_personen
AFTER UPDATE ON personen
BEGIN
    -- Aktionen, kann aber kein :new / :old verwenden
END;
```

ID	Name	Abteilung	Matrikel
1	Meier	Informatik	1234
2	Müller	Wirtschaft	5678
3	Bauer	Informatik	1133
4	Schulz	Wirtschaft	2244

Mutating Table Problem

Mutating Table Problem

```
create or replace
TRIGGER trg_personen
AFTER UPDATE ON personen
FOR EACH ROW
DECLARE cnt NUMBER;
BEGIN
  SELECT COUNT(*) INTO cnt FROM PERSONEN
  WHERE Matrikel LIKE 'I-%';
END;
```

ID	Name	Abteilung	Matrikel
1	Meier	Informatik	I-1234
2	Müller	Wirtschaft	5678
3	Bauer	Informatik	1133
4	Schulz	Wirtschaft	2244

Trigger: „Mutating Tables“

- Welchen Zustand der Datenbank sieht ein zeilenbasierter Trigger?
- Ggf. ist in einem Update-Trigger ein Teil der Daten schon geändert.
- Daher Einschränkung bei Oracle:
  - Ein **zeilenbasierter Trigger** darf die Zieltabelle nicht verwenden
  - Ansonsten: „ORA-04091: Tabelle PERSONEN wird gerade geändert, Trigger/Funktion sieht dies möglicherweise nicht“
  - Engl: „Table is mutating“
- Lösung:
  - Zugriff auf die Tabelle im Statement-Trigger

Beispiel: Trigger 1

„Überprüfung, ob das Gesamtbudget einer Abteilung immer größer ist als das Budget all ihrer Projekte“

```
CREATE OR REPLACE TRIGGER trg_check_budget
    AFTER INSERT OR UPDATE OF budget, abtnr ON Projekt
DECLARE
    v_Cnt NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_Cnt
    FROM Abteilung a
    WHERE budget < (SELECT SUM(budget)
                    FROM Projekt
                    WHERE abtnr = a.abtnr);
    IF v_Cnt > 0 THEN
        RAISE_APPLICATION_ERROR (
            num => -20000, msg => 'Unzulässiges Budget'
        );
    END IF;
END;
```

Beispiel: Trigger 2

Protokollierung des Gehaltes bzw. seiner Änderung nach Änderungen (insert, update) eines Angestellten.

```
CREATE OR REPLACE TRIGGER trg_gehaltsaenderungen
  AFTER INSERT OR UPDATE ON Angestellte
  FOR EACH ROW
  WHEN (nvl(new.gehalt,0) <> nvl(old.gehalt,0))
DECLARE
  diff NUMBER;
BEGIN
  diff  :=  :new.gehalt - :old.gehalt;
  INSERT INTO protocol_gehaltsaenderungen
    (id, anr, altes_gehalt,
     neues_gehalt, differenz, geaendert, datum)
  VALUES
    (seq_gehaltsaenderungen.nextval, :new.anr, :old.gehalt,
     :new.gehalt, diff, (select user from dual), sysdate);
END;
```

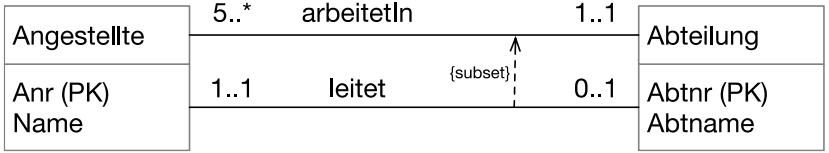
Beispiel: Trigger 2 (Version PostgreSQL)

Protokollierung der Gehaltsänderungen nach Änderungen eines Angestellten.

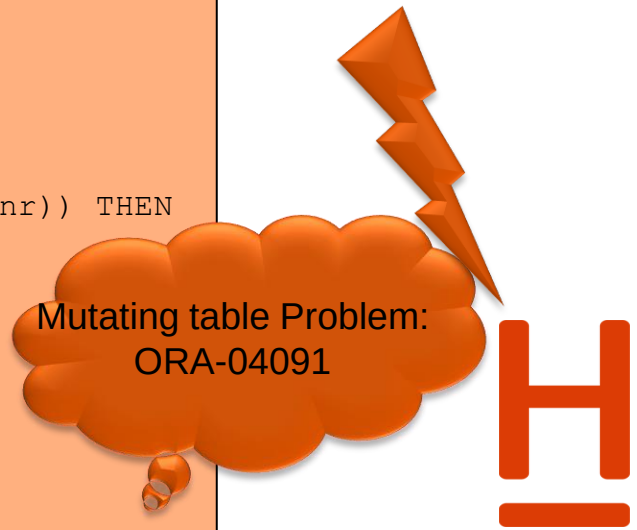
```
CREATE OR REPLACE FUNCTION trg_prot_gehalt_aend() RETURNS trigger AS $protgehalt$
begin
    INSERT INTO protocol_gehaltsaenderungen
    (id, anr, altes_gehalt, neues_gehalt, differenz, geaendert, time)
    VALUES
        (nextval('seq_gehaltsaenderungen'), NEW.anr, OLD.gehalt,
         NEW.gehalt, NEW.gehalt-OLD.gehalt, current_user, current_timestamp);
    RAISE NOTICE 'Trigger: Gehaltsaenderung vermerkt';
    RETURN NEW;
end;
$protgehalt$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER trg_gehaltsaenderungen
  AFTER UPDATE ON Angestellte  -- INSERT geht nicht mit OLD
  FOR EACH ROW
WHEN (coalesce(NEW.gehalt,0) <> coalesce(OLD.gehalt,0))
EXECUTE FUNCTION trg_prot_gehalt_aend();
```

Integritätsbedingung

Anzahl der Angestellten einer Abteilung >= 5

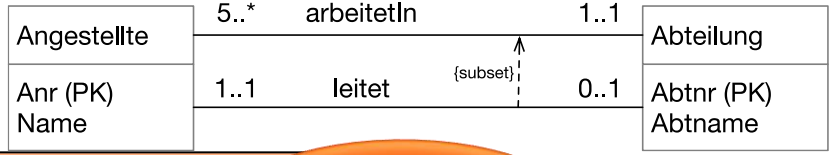



```
CREATE OR REPLACE TRIGGER trg_Angest5plus
AFTER DELETE OR UPDATE OF abtnr ON Angestellte
FOR EACH ROW
DECLARE anzAngest NUMBER;
BEGIN
    IF (DELETING OR (UPDATING AND :old.abtnr <> :new.abtnr)) THEN
        SELECT COUNT(*) INTO anzAngest
        FROM Angestellte WHERE abt = :old.abt;
        IF (anzAngest < 5) THEN
            RAISE_APPLICATION_ERROR(num => -20000,
                msg => 'Angest5Plus verletzt!' );
        END IF;
    END IF;
END;
```



Integritätsbedingung

Anzahl der Angestellten einer Abteilung >= 5

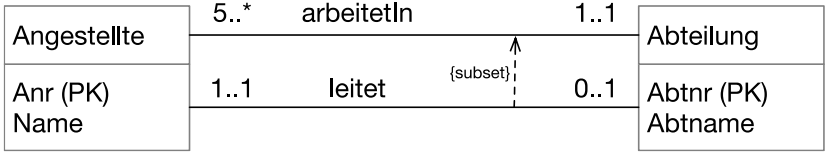



```
CREATE OR REPLACE TRIGGER trg_Angest5plus
AFTER DELETE OR UPDATE OF abtnr ON Angestellte

DECLARE anzAngest NUMBER;
BEGIN
    SELECT min(cnt) INTO anzAngest FROM
        (SELECT COUNT(*) cnt, abtnr
         FROM Angestellte GROUP BY abtnr);
    IF (anzAngest < 5) THEN
        RAISE_APPLICATION_ERROR(num => -20000,
                                msg => 'Angest5Plus verletzt!'
        );
    END IF;
END;
```

Kein Zeilen-Trigger mehr

Kein :new/:old, dafür gesamte Tabelle nutzen

Integritätsbedingung
Durchsetzung der Teilmengen-Bedingung




```
CREATE OR REPLACE
TRIGGER trg_AbtTeilmenge
AFTER UPDATE OF leitung ON Abteilung FOR EACH ROW
DECLARE v_abtnr number;
BEGIN
   IF (:new.leitung <> :old.leitung) THEN
      SELECT abtnr INTO v_abtnr
      FROM Angestellte WHERE anr = :new.leitung;
      IF v_abtnr != :new.abtnr THEN
         RAISE_APPLICATION_ERROR(num => -20000,
                                 msg => 'Teilmenge verletzt');
      END IF;
   END IF;
END;
```

Nur für Update auf Abteilung, nicht auf Angestellte!!!

Trigger – Weitere Anwendungen

- (transitionale) Integritätsregeln
  - z.B. Gehälter dürfen nicht sinken
- Nachführen redundanter (aggregierter) Daten
  - z.B. Gehaltskosten pro Abteilung
    oder Budgets der Projekte einer Abteilung
- Protokollierung
- Workflow-Steuerung
  - Versenden von Nachrichten an Benutzer
- Anstoßen externer Aktionen
- Berechtigungsüberprüfungen

Zusammenfassung

Aktive Datenbanksysteme

- Ein Datenbanksystem ist *aktiv,* wenn es auf (externe oder interne) *Ereignisse* durch (externe oder interne) *Aktionen* reagiert.
- Durch eine Regelbasis wird festgelegt, auf welche Ereignisse wie reagiert werden soll
- Der effektive Programmablauf wird dadurch dynamisch verändert
- Der Programmablauf ist ggf. nicht deterministisch
- Aufwand für Ausführung aller ausgelösten Trigger entsteht in der auslösenden Aktion → sollte nicht zu aufwändig sein
- Kaskadierende Trigger ggfs bedenken
- Kann bei Nichtkenntnis der Trigger zu Verwirrung bei den Ergebnissen führen

Zusammenfassung

SQL-Trigger am Beispiel Oracle

- Syntax; Verwendung von :new / :old
- Zeilen- und Statementtrigger
- Mutating Table Problem

Beispiele für Trigger-Anwendungen

- Integritätsregeln
- Protokollierung