Kap. 3: DB-interne Programmierung

a) Stored Procedures mit PL/SQL

Statisches SQL: - SQL in bekannte Sprache (z.B. C oder Java) einbetten
Andere Herangehensweise:

- SQL mit prozeduralen Elementen erweitern - PL/SQL (SQL-Erweiterung von Oracle)
- 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“?

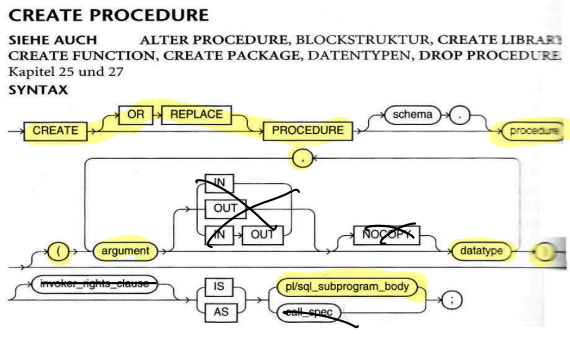
- **Stored Procedures** sind alleinstehende Einheiten, die (in einer Datenbank) global definiert und verfügbar sind
- **Stored Procedures** lassen sich in proprietären Datenbanksprachen (z. B. PL/SQL) definieren und in der Datenbank ablegen
- Als Stored Procedures werden in der Regel **Prozeduren** (ohne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

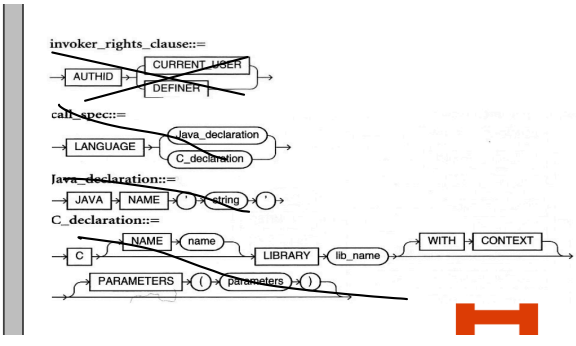
Beispiel: Arbeitszeitenverbuchung

```
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;
```

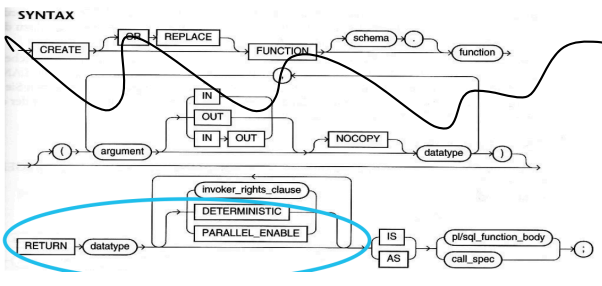
- Aufruf mit `execute day_finished(1, 9);`

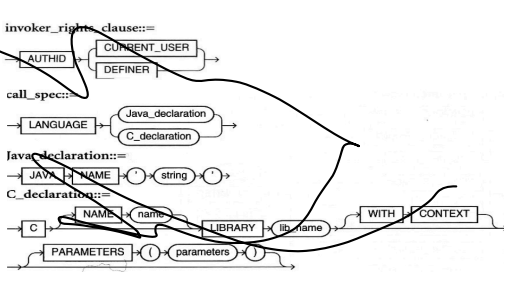
Stored Procedures – Syntax Prozedur





Stored Procedures – Syntax Funktion





PL/SQL: Aufbau

DECLARE {
⋮
BEGIN {
⋮
EXCEPTION {
⋮
END {
⋮
}

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

PL/SQL: Variablen und Datentypen

Variablen werden im DECLARE-Teil definiert

- Syntax: `variable_nametype [CONSTANT] [NOTNULL] [:=value];`

vorhandene Datentypen sind

- Skalare Typen ,z.B. `DATE, NUMBER(3)`
- Zusammengesetzte Typen `(RECORD, TABLE, VARRAY)`
- Verweise auf Cursor `(REFCURSOR)`
- ...

identischen Datentypen
nicht kompatibel

Mit %TYPE kann der Datentyp einer Datenbankspalte referenziert werden

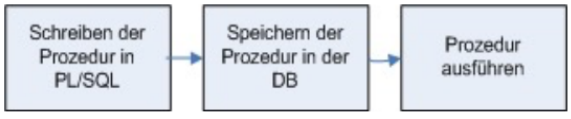
- z.B. `students.first_name%TYPE` code passt sich dem Datentyp an

PL/SQL: Ausführung

Stored Procedures lassen sich in verschiedenen Formen ausführen:

- **interaktiv** mit SQLDeveloper
  - EXECUTE prozedurname (parameterwerte);
- (**innerhalb von PL/SQL-Blöcken** wie normale Prozeduren aufrufbar)
- in **SQL-Anweisungen** (nur Funktionen)
- ~~In Pro*C~~
  - ~~über EXEC SQL EXECUTE mit aktuellen Parametern versehen und starten (wie normale Prozeduren)~~
- ~~Über **JDBC** aus Java heraus~~

PL/SQL: Ablauf



1. Schritt: **Schreiben** der Prozedur bzw. der Funktion in PL/SQL
   - 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
   - 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;
```

Aufruf einer "Stored Function"

Über DUAL:

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

In beliebigem SQL-Select:

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

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

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

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

Zusammenfassung Teil a)

- "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
- Unser Beispiel: PL/SQL von Oracle
- Die Sprache 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, inkl. NULL

- Information Hiding

  VT: Man muss sich nicht mit der Implementierung auseinander setzen
  NT: Detail Informationen werden versteckt

- Änderungen am Datenmodell / Fehlerbehandlung, Konsistenz

  VT: Es gibt eine Schema Prüfung durch %TYPE

- Rechenzeit / Speicherbedarf auf Clients

  VT: Mehr Last auf der Server als auf den Clients

- Sicherheit

  VT: Zugriffe werden kontrolliert?

- Benötigtes Know-How bei Entwicklern

  VT: Beim Ausführen braucht man kein Know-how
  NT: Beim Entwickeln der SP schon?

- Schichtentrennung

  VT: kann eine Schicht ablösen?

- Wartung der Anwendung

  NT: Wenn Know-how fehlt?

eher weniger lernen

b) Stored Procedure mit Java

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)

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

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

Java Stored Procedures - Bewertung

Vorteile

- Alle Vorteile von SPs generell
  - Z.B. Performance etc.
- Java wird als einzige und moderne Sprache verwendet
  - Flexibilität: Transformation zu anderen Mechanismen (RMI, Servlets) möglich

Nachteile

- Verwendung etwas umständlich (Wrapper)
- Keine „echte“ Objektorientierung
- 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

Aktive Datenbanksysteme

Extern: außerhalb der Datenbank, wie externe Methode aufrufen um Mail zu versenden

Weitere Lösung: Verwendung eines aktiven Datenbanksystems!

Was ist ein aktives Datenbanksystem?

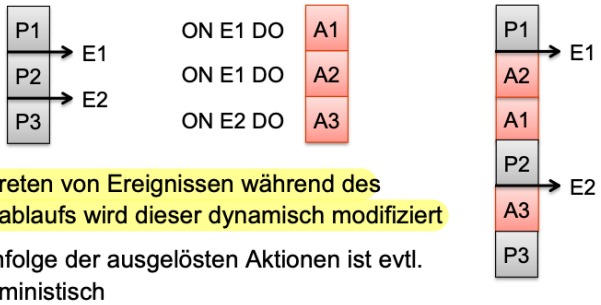
> Ein Datenbanksystem ist *aktiv*, wenn es auf (externe oder interne) *Ereignisse* durch (externe oder interne) *Aktionen* reagiert.

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

- 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

- 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)

Modifikation des Programmablaufs:



Durch Eintreten von Ereignissen während des Programmablaufs wird dieser dynamisch modifiziert

Die Reihenfolge der ausgelösten Aktionen ist evtl. nicht deterministisch

Vorteile

- Anwendungsprogrammierung wird entlastet
- Weniger Wissen im Code, mehr Wissen in der Datenbank
- Erhöhte Flexibilität, z.B. im Fehlerfall
- Modifikation der Ablaufbeschreibung
- Modifikation der Regelbasis Flexibilität
- Flexible Verknüpfung von Prozessschritten über aktive Regeln

Nachteile:

- Nicht-Determinismus,
- Ablauf der Entwicklung bzgl. DB-Code (git),
- Perfomance,
- unerwartete Aufrufe (Schleifen => Abstürze),
- Komplexität steigt mehr Wissen in der Datenbank & Fehlersuche,
- Fehlerbehandlung schwieriger

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

Trigger

- Trigger können gerollbacked werden
- 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 ist im Allgemeinen nicht-deterministisch.

ON .. IF .. DO ..

unterschiedliche Syntax

```
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;
```

```
<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
```

- **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.

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, ...`

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

- Nur Zeilen-Trigger können auf :new & :old zugreifen, Statement-Trigger können das nicht

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

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

- Auf die zu veränderte Tabelle darf nicht im Trigger zugegriffen werden, da nicht klar ist was für Werte diese haben kann und für jede Zeile unterschiedlich ist wie das Beispiel unten demonstriert. Der COUNT(*) kann für jede Zeile unterschiedlich sein.

```
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

„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;
```

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

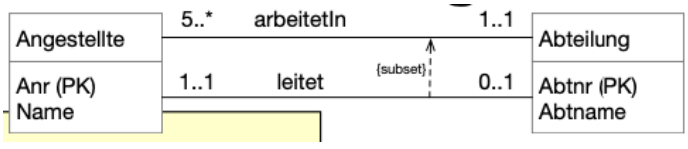
```
CREATE OR REPLACE TRIGGER trg_gehaltsaenderungen
  AFTER INSERT OR UPDATE ON Angestellte
  FOR EACH ROW
  WHEN (nvl(new.gehalt,0) <> nvl(old.gehalt,0))
DECLARE
  diff NUMBER;
BEGIN
  diff  :=  :new.gehalt - :old.gehalt;
  INSERT INTO gehaltsaenderungen
    (id, anr, altes_gehalt,
     neues_gehalt,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

Integritätsbedingung Anzahl der Angestellten einer Abteilung >= 5

- Hier Mutating Table Problem, deshalb auf Statement-Trigger umwandeln

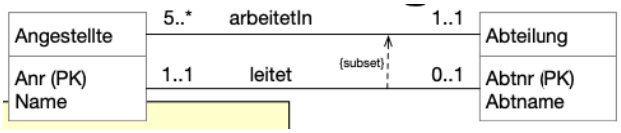
```
CREATE OR REPLACE TRIGGER trg_Angest5plus
AFTER DELETE OR UPDATE OF abtnr ON Angestellte
FOR EACH ROW
DECLARE anzAngest NUMBER;
BEGIN
    IF (DELETING OR (UPDATING AND :old.abtnr <> :new.abtnr)) THEN
        SELECT COUNT(*) INTO anzAngest
        FROM Angestellte WHERE abt = :old.abt;
        IF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

```
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;
```



Integritätsbedingung Durchsetzung der Teilmengen-Bedingung

```
CREATE OR REPLACE
TRIGGER trg_AbtTeilmenge
BEFORE 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;
```



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 (externeoderinterne) 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. nichtdeterministisch

SQL-Trigger am Beispiel Oracle

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

Beispiele für Trigger-Anwendungen

- Integritätsregeln
- Protokollierung