ORACLE9i:

Triggers

***Autoren**:* *Stefan Berger, Christian Eichinger*

Günter Preuner, Thomas Thalhammer

in Anlehnung an “Oracle 7: Triggers”
Folien von Stefan Rausch-Schott und Werner Retschitzegger,
Abteilung für Informationssysteme, Uni Linz, 1997

Aktive Datenbanksysteme
Motivation

Triggers realisieren die Idee aktiver Datenbanksysteme

(“Regeln” als separate Komponenten) für relationale Datenbanksysteme.

Prinzipielle Regelstruktur (Event-Condition-Action - ECA):

ON *Ereignis*

IF *Bedingung*

DO *Aktion*

- kein *unmittelbarer* Zusammenhang zwischen Benutzerinteraktion und Ausführung des Triggers (nur mehr *mittelbar* aufgrund der von einem Benutzer ausgelösten bzw. spezifizierten Ereignisse)

Motivation
Beispiel

„Reorder Trigger“

Triggering Statement (Ereignis)

```
AFTER UPDATE OF parts_on_hand ON inventory
FOR EACH ROW
```

Trigger Restriction (Bedingung)

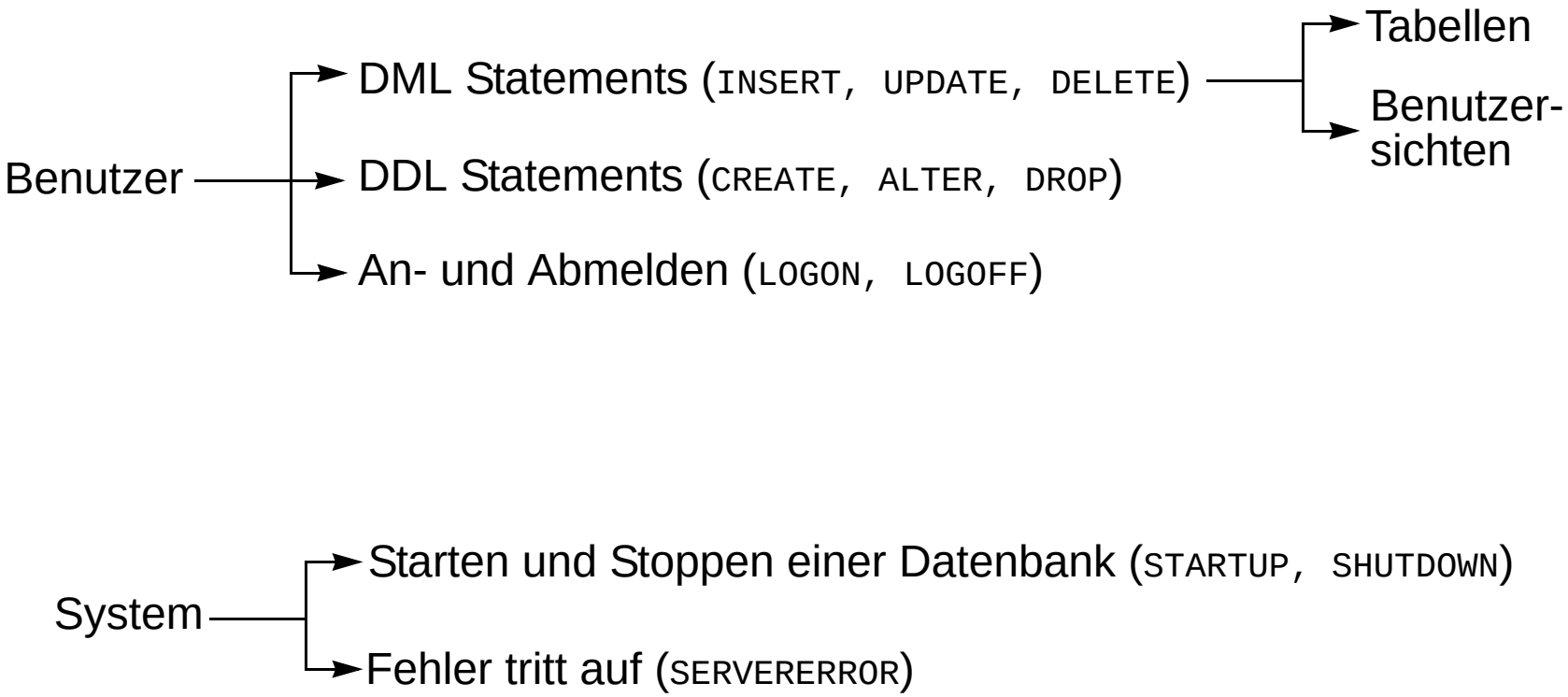
```
WHEN (new.parts_on_hand < new.reorder_point)
```

Trigger Action

```
DECLARE                          /* a dummy variable for counting */
 x NUMBER;
BEGIN
 SELECT COUNT(*) INTO x          /* query to find out if part has already been */
 FROM pending_orders             /* reordered - if yes, x=1, if no, x=0 */
 WHERE part_no= :new.part_no;

 IF x=0
 THEN                            /* part has not been reordered yet, so reorder */
  INSERT INTO pending_orders
  VALUES (:new.part_no, :new.reorder_quantity, sysdate);
 END IF;                         /* part has already been reordered */
END;
```

Trigger-Komponenten
Überblick über Ereignisse



DML-Triggers auf Tabellen
Ereignis

- spezifiziert **SQL-Statement-Typ**, der den Trigger feuert - `INSERT, DELETE, UPDATE`
  - **disjunktive** Verknüpfung möglich
  - in diesem Fall können **konditionale Prädikate** `INSERTING, UPDATING, DELETING` den Ereignis-Typ herausfinden u. in Abhängigkeit davon passenden Code ausführen
  - bei **UPDATE-Ereignis** kann eine **Attributliste** angegeben werden
- und die damit verbundene **Tabelle** - genau eine
  - **nicht** auf **Views** definierbar
  - Trigger, die auf Basis-Tabellen einer View definiert sind, werden jedoch auch **bei Operationen auf die View gefeuert**

DML-Triggers auf Tabellen
Bedingung

- `WHEN (new.empno > 0)`
  - **bei Row-Triggern optionale** Spezifikation eines **logischen Ausdrucks** (WHEN-Klausel) der wahr sein muss, damit Trigger feuert
  - meist verwendet für den Vergleich von **Spalten-Werten**
  - **erlaubte Ausdrücke**: SQL-Funktionen mit booleschem Ergebnis
  - **nicht erlaubt**: PL/SQL u. Subqueries

DML-Triggers auf Tabellen
Aktion

- Variante 1 - PL/SQL-Block
  - **erlaubte Ausdrücke:** SQL u. PL/SQL-Befehle u. Aufruf gespeicherter Proz. innerhalb von `begin ... end;`
  - **nicht erlaubt**: DDL-Statements, Statements zur Transaktionskontrolle, keine Variablen-Deklarationen als **LONG und LONG RAW Datentypen**
- Variante 2 - CALL
  - Anstelle eines PL/SQL-Blocks (`begin ... end;`) wird eine einzelne Prozedur mit
    `CALL [benutzer.][paket.]prozedurname (parameter)`
    aufgerufen
  - Dieses Statement darf NICHT durch ein `;` abgeschlossen werden

DML-Triggers auf Tabellen
Trigger-Typen

- Trigger-Typ spezifiziert, **wie oft die Trigger-Bedingung/Aktion ausgeführt** werden soll
- **Row-Trigger** `FOR EACH ROW`
  - **1mal pro** vom triggernden Ereignis betroffener **Zeile**
  - sinnvoll, wenn Trigger-Aktion von Daten, die das triggernde Ereignis bei der Verarbeitung der einzelnen Zeilen zur Verfügung stellt, abhängt
    - Auditing
- **Statement-Trigger** `FOR EACH STATEMENT`
  - **1mal pro** triggerndem **Ereignis**, egal wieviele Zeilen bearbeitet werden
    - Zugriffsschutz

DML-Triggers auf Tabellen
Trigger-Typen

- **BEFORE**- u. **AFTER** spezifiziert, **wann** (vor oder nach triggerndem Statement) **der Trigger ausgeführt** werden soll

		BEFORE	AFTER
UPDATE	row	ja	ja
	statement	ja	ja
DELETE	row	ja	ja
	statement	ja	ja
INSERT	row	ja	ja
	statement	ja	ja

- **pro Tabelle** u. **Trigger-Typ** sind ***beliebig viele*** **Triggers** erlaubt

- Anwendung von BEFORE-Triggern:
  - Unnötigen **Verarbeitungsaufwand** verhindern, da Trigger-Aktion bestimmt, ob das triggernde Statement weiter ausgeführt wird - Rollback bei Exception
  - **ableiten** von **Spalten-Werten** vor Ausführung des triggernden Statements

DML-Triggers auf Tabellen
Zugriff auf alte/neue Spaltenwerte

- nur bei **Row**-Triggern
- Statements in Bedingung u. Aktion haben über 2 Korrelationsnamen „**OLD**“ u. „**NEW**“ Zugriff auf alte u. neue Spalten-Werte der gerade bearbeiteten Zeile

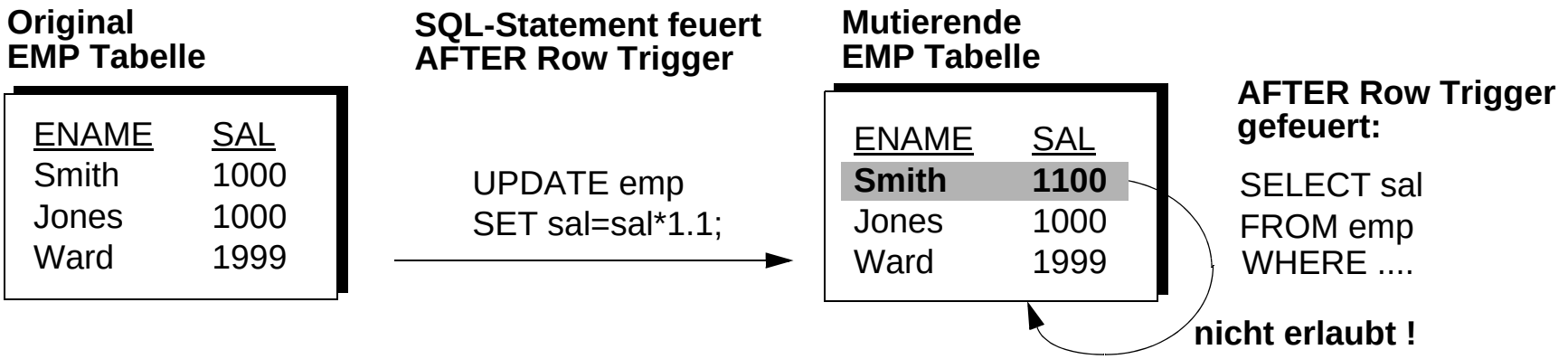
  `IF :new.sal > 1000 ... oder IF :new.sal < :old.sal`
- **Ereignisart** bestimmt **Sinnhaftigkeit** der Korrelationsnamen:

		Korrelationsnamen	
		OLD	NEW
Ereignisart	INSERT	nein	ja
	UPDATE	ja	ja
	DELETE	ja	nein

- Syntax in Bedingung: `new.column, old.column`
  Syntax in Aktion: `:new.column, :old.column`

DML-Triggers auf Tabellen
Mutierende Tabellen

- „Mutierende Tabelle“ = Tabelle, die **gerade modifiziert** wird
- Beispiel:



  - Befehle eines **Row-Triggers** dürfen mutierende Tabelle weder **lesen noch schreiben**
  - verhindert potentielle Endlosschleifen

DML-Triggers auf Tabellen
Definition von Triggern

- **Trigger-Namen** müssen in Bezug auf andere Trigger **eindeutig** sein (nicht bezüglich anderer Schema-Objekte)
- Beim Löschen einer Tabelle werden automatisch alle abhängigen Trigger gelöscht
- ein Trigger kann **nicht explizit verändert** werden
  - Neudefinition über `OR REPLACE` Option im `CREATE TRIGGER` Befehl
  - oder löschen mit `DROP TRIGGER` und erzeugen über `CREATE TRIGGER`
- Trigger können **aktiviert** (`ENABLE`) = default und **deaktiviert** (`DISABLE`) werden

```
ALTER TRIGGER reorder DISABLE;
ALTER TABLE inventory DISABLE ALL TRIGGERS;
```

DML-Triggers auf Tabellen
Beispiel

```
CREATE OR REPLACE TRIGGER dummy
BEFORE INSERT OR UPDATE OF sal,ename ON emp
FOR EACH ROW
WHEN (new.empno > 0)
DECLARE
    /* variables, constants, coursors, etc. */
BEGIN
    /* PL/SQL block */
    IF UPDATING('sal') THEN ... END IF; ...
EXCEPTION
    /* exception handling */
END;
```

Potentielle triggernde Ereignisse:

```
INSERT INTO emp VALUES(...);
INSERT INTO emp SELECT ... FROM ...;
UPDATE emp SET sal=sal+100;
```

Triggers auf Tabellen
Anwendungsbereiche

- Automatisches **Ableiten von Spalten-Werten**
- Verhindern invalider **Transaktionen**
- **Komplexe Autorisierungen**
- **Referentielle Integrität** (über Knoten einer **verteilten DB)**
- Synchronisieren von **Tabellen-Replikaten**
- Komplexe „**Business Rules**“
- Transparentes **Ereignis-Logging**
- **Statistiken** über Tabellen-Zugriff
- Komplexes **Auditing**

DML Triggers auf Benutzersichten (Views)
Ereignis/Bedingung/Aktion

- **INSTEAD OF** (ersetzt BEFORE und AFTER): Trigger wird *an Stelle* der den Trigger auslösenden DML Operation durchgeführt

- Nur **Row-Triggers** erlaubt (`for each row` gilt implizit)

  - **Ereignis-Deklaration:**
    ```
    create or replace trigger <name>
    instead of delete | insert | update
    ```
    (Anmerkung: `OF`-Klausel bei `update`-Ereignis ist NICHT erlaubt)

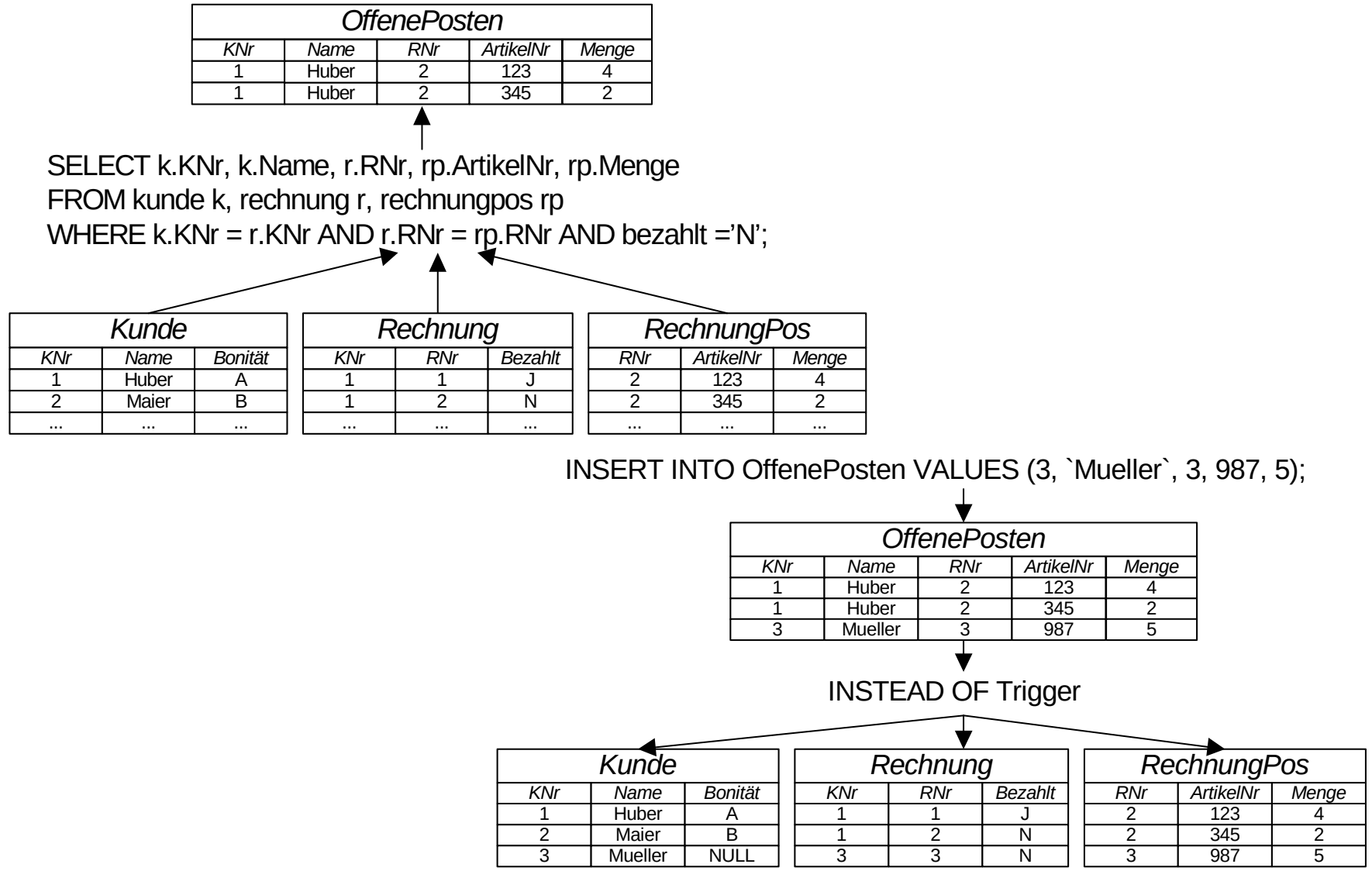
  - Bedingung: Keine `WHEN`-Klausel erlaubt

  - Aktion ist gleich wie bei DML Triggers auf Tabellen

- Benutzersichten sind nicht ‘‘mutierend’’

- **Anwendungsmöglichkeiten:** Updates über Benutzersichten, die aus Join entstehen; statt des (unzulässigen) Updates über die Benutzersicht werden vom Trigger die Basisrelationen verändert

DML Triggers auf Benutzersichten (Views)
Beispiel



DML-Trigger
Beispiel

- Gegeben sei das aus der Übungseinheit PL/SQL bereits bekannte Schema zur Verwaltung von Mitarbeiter-Daten:

```
CREATE TABLE mitarbeiter (
   svnr NUMBER(10) PRIMARY KEY NOT NULL,
   name VARCHAR2(64) NOT NULL,
   abteilung NUMBER(8) NOT NULL,
   tel NUMBER(5) NOT NULL,
   gehalt NUMBER(8,2) NOT NULL,
   CONSTRAINT ma_fk FOREIGN KEY (abteilung) REFERENCES
abteilung(abt_nr)
);

CREATE TABLE abteilung (
   abt_nr NUMBER(8) PRIMARY KEY NOT NULL,
   bezeichnung VARCHAR2(32) NOT NULL
);
```

DML-Trigger
Beispiel (Vortsetzung)

- Durch einen Trigger sollen nachfolgende Einschränkungen bei Änderungen am gehalt-Attribut eines Mitarbeiters erzwungen werden:

  - Beim Einfügen eines neuen mitarbeiter-Tupels darf das Gehalt nicht kleiner als EUR 300 sein.

  - Wird das bestehende Gehalt eines bereits gespeicherten Mitarbeiters geändert, so darf dies ausschließlich vom Geschäftsführer (Benutzer 'boss') durchgeführt werden. Gehaltserhöhungen dürfen von allen BenutzerInnen der Datenbank durchgeführt werden. (Benutzer können über die Systemvariable USER abgefragt werden)

DDL Triggers
Ereignis

- DDL Triggers dienen zum Behandeln (Protokollieren, ...) von Schemaänderungen und für komplexe Autorisierungsprozeduren in einer Datenbank bzw. in einem Benutzerschema

- Nur **Row-Triggers** erlaubt (`for each row` gilt implizit)

  - **Timing** $\in$ {`BEFORE, AFTER`}

  - **DDL-Ereignisse** $\in$ {`CREATE, ALTER, DROP, DDL, REVOKE, GRANT, ...`}[1]

  - Ereigniseintritt wird entweder innerhalb eines bestimmten Schemas oder in allen zu einer Datenbank gehörenden Schemata signalisiert[2]

  - **Deklaration:**

```
create or replace trigger (before | after)
(create | alter | drop)
```

1. Oracle schränkt diese Ereignisse auf folgende Datenbankobjekte ein: CLUSTER, FUNCTION, INDEX, PACKAGE, PROCEDURE, ROLE, SEQUENCE, SYNONYM, TABLE, TABLESPACE, TRIGGER, TYPE, VIEW, USER
2. Um einen DATABASE-Trigger zu definieren, benötigt man das Systemprivileg ADMINISTER DATABASE TRIGGER

DDL Triggers
Bedingung/Aktion

- In Bedingung und Aktion kann auf verschiedene Attribute des Ereignisses zugegriffen werden (Auszug):
  - `ora_sysevent` ... Art des Ereignisses (z.B. `CREATE`)
  - `ora_dict_obj_type` ... Typ des Schema-Objekts (z. B. `TABLE`)
  - `ora_dict_obj_name` ... Name des Schema-Objekts
  - `ora_login_user` ... Benutzerkennung des Ereignisauslösers
- Beispiel:

```
create or replace trigger ddl_scott after ddl on database
 when (ora_login_user = 'SCOTT')
 begin
   insert into events values (ora_sysevent || ' ' ||
     ora_dict_obj_type || ' '|| ora_login_user || '.' ||
     ora_dict_obj_name, sysdate);
 end;
```

An- und Abmelden
Ereignis/Bedingung/Aktion

- dienen zur Ausführung von Aktionen *unmittelbar nach dem Anmelden* am System bzw. *unmittelbar vor dem Abmelden* vom System

- Nur **Row-Triggers** erlaubt (`for each row` gilt implizit)

  - Kombinationen `AFTER LOGON` und `BEFORE LOGOFF` erlaubt

  - Nach Ereigniseintritt wird entweder nur innerhalb ein bestimmtes Schemas oder in allen zu einer Datenbank gehörenden Schemata gesucht

  - Ereignisattribute = {`ora_sysevent`, `ora_login_user`, `ora_instance_num`, `ora_database_name`, `ora_client_ip_address`[1]}

- Beispiel:

```
create or replace trigger deny_logon AFTER LOGON on SCHEMA
when (ora_client_ip_address = '140.78.58.26')
begin .... end;
```

1. `ora_client_ip_address` ist nur für LOGON-Triggers verfügbar.

Starten und Stoppen einer Datenbank
Ereignis/Bedingung/Aktion

- dienen zur Ausführung von Aktionen *beim Starten* einer Datenbankinstanz bzw. *beim Stoppen* einer Datenbankinstanz
- Nur **Row-Triggers** erlaubt (`for each row` gilt implizit)
  - **Ereignisdeklaration** `AFTER STARTUP ON DATABASE` oder `BEFORE SHUTDOWN ON DATABASE`
  - **Ereignisattribute** = {`ora_sysevent`, `ora_login_user`, `ora_instance_num`, `ora_database_name`}
- **Besonderheit**: Diese Trigger werden in einer separaten Transaktion ausgeführt.

Fehlereintritt
Event/Condition/Action

- dienen zur asynchronen Fehlerbehandlung
- Nur **Row-Triggers** erlaubt (`for each row` gilt implizit)
  - **Ereignisdeklaration** `AFTER SERVERERROR ON (DATABASE | SCHEMA)`
  - **Ereignisattribute** = {`ora_sysevent`, `ora_login_user`, `ora_instance_num`, `ora_database_name`, `ora_server_error(...)`[1], `ora_is_servererror(...)`[2]}
  - Optionale **Bedingung**
- **Besonderheit**: Diese Trigger werden in einer separaten Transaktion ausgeführt.

1. Parameter ist die Position am Errorstack (1 bedeutet oberste Fehlermeldung, dh. die letzte Fehlermeldung die auf den Errorstack gelegt wurde)
2. Parameter ist ein numerischer Fehlercode (z.B. der Fehler ORA-1020 hat den numerischen Fehlercode 1020)

Ausführungsmodell
Multiple Trigger

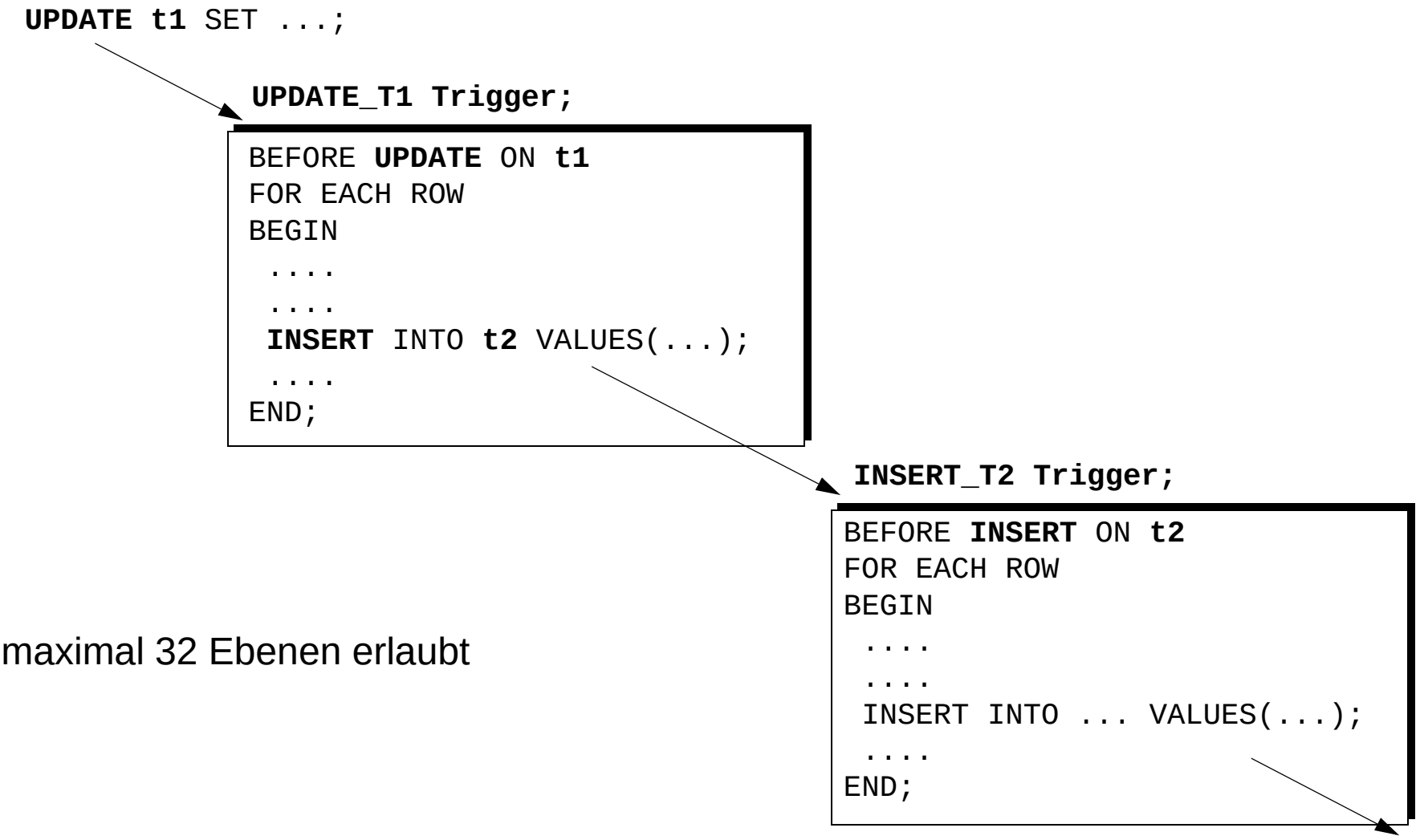
- Ausführungsmodell zur Bestimmung der **Ausführungs-Sequenz**:

```
1. Execute BEFORE statement trigger.
2. Loop for each row affected by the SQL statement.
    a. Execute BEFORE row trigger
    b. Lock and change row, and perform integrity constraint checking.
        (The lock is not released until the transaction is committed.)
    c. Execute AFTER row trigger.
3. Complete deferred integrity constraint checking.
4. Execute AFTER statement trigger.
```

- **alle** aufgrund eines SQL-Statements durchgeführten Aktionen u. Checks müssen **erfolgreich sein**

- tritt eine **Exception** auf u. wird sie nicht explizit behandelt, wird auf alle **Aktionen** u. das **triggernde Statement** selbst ein **Rollback** durchgeführt

Ausführungsmodell
Kaskadierende Trigger



maximal 32 Ebenen erlaubt

Triggers
Fehlersituation

- Falls bei der Ausführung eines Triggers ein Fehler auftritt und dadurch die Ausführung dieses Triggers abgebrochen wird, gilt das auslösende Ereignis als nicht ausgeführt.

- Dies kann dazu führen, dass ein Fehler in einem LOGIN-Trigger ein Anmelden an der Datenbank verhindert bzw. ein Fehler in einem LOGOFF-Trigger ein Abmelden von der Datenbank verhindert.

- Einzige Ausnahme: STARTUP und SHUTDOWN sind auch dann erfolgreich, wenn ein von diesen Ereignissen gefeuerter Trigger einen Fehler verursacht.

Ausführungsmodell
Speicherung und Ausführung der Trigger-Aktion

- Triggers werden so wie “Stored Procedures“ **einmal** kompiliert und dann als P-Code (Pseudo-Code, PL/SQL in kompilierter Form) in der Datenbank verwaltet
  - ältere Versionen von Oracle haben nur den Quellcode gespeichert
  - jedesmal wenn der Trigger gefeuert wurde und nicht im ‘‘shared pool’’ geladen war, musste die Trigger-Aktion zuerst geladen und kompiliert werden (anonymer PL/SQL-Block)
- die Ausführung eines Triggers ist ebenfalls gleich wie die Ausführung einer “Stored Procedure”.
  - **Einziger Unterschied**: Die Berechtigung einen Trigger auszuführen hängt von der Berechtigung ab, das triggernde SQL-Statement auszuführen.