Kapitel 6

Integritätssicherung

Begriffsverwirrung “Integrität”

1. “semantische Integrität”

   Problem: Erhaltung der logischen Korrektheit (Konsistenz) bei DB-Änderungen durch (berechtigte) Benutzer

   Spektrum: Plausibilitätstests . . . Kontrollregeln (abhängig von Aufwand und Relevanz) ⟶ **Integrität(ssicherung)** “integrity”

2. “Zugriffsintegrität”

   Problem: Ausschluß von DB-Zugriffen durch (dazu) unberechtigte Benutzer ⟶ **Datenschutz** “security”

3. “operationale” oder “Ablauf-Integrität”

   Problem: konkurrierende DB-Zugriffe mehrerer Benutzer ⟶ **Synchronisation** “concurrency control”

4. “physische Integrität”

   Problem: Wiederherstellung zerstörter Daten nach Zerstörung durch technische Fehler ⟶ **Datensicherung** “recovery”

6.1 Integritätsbedingungen

Hier geht es um semantische Integrität. Zwei Klassifikationen:

Def.: **Integritätsbedingungen (IBen, "integrity constraints")** heißen **(modell-)inhärent** oder **strukturell**, wenn sie sich durch Strukturen bzw. Konzepte des gewählten Datenmodells (z.B. durch ER-Diagramme)[1] ausdrücken lassen; **explizite** IBen sind zusätzlich zu spezifizieren.

Def.: Integritätsbedingungen können betreffen:

(a) (einzelne) DB-Zustände σ $\longrightarrow$ **statische IBen**

(b) DB-Änderungen, also Zustandsübergänge
$\langle \sigma_{\text{old}}, \sigma_{\text{new}} \rangle$ $\longrightarrow$ **dynamische (transitionale) IBen**

(c) ganze DB-Entwicklungen, also Zustandsfolgen
$\langle \sigma_{\text{initial}}, \sigma_1, \sigma_2, \ldots, \sigma_{\text{aktuell}}, \ldots \rangle$ $\longrightarrow$ **dynamische (temporale) IBen**

DB-Zustände/Zustandsübergänge/-folgen, in denen alle über einem DB-Schema spezifizierten statischen/transitionalen/temporalen IBen gültig sind, heißen **zulässig**.

[1] Dem Relationenmodell "inhärente" IBen sind Angaben zu Datentypen, Schlüsseln, Nullwerten, Eindeutigkeiten, Referenzen und Fremdschlüsseln (vgl. Kap. 2). In einem relationalen DBMS sollten mindestens diese "eingebaut" sein.

Einheiten der Integritätsüberwachung

- Inhärente IBen sollten von jeder Änderungsoperation (**insert**/**delete**/**update**) erhalten werden.
- Bei manchen expliziten IBen ist es wünschenswert, sie erst nach mehreren DB-Operationen zu prüfen, z. B.

  Zu jeder Bestellung gibt es einen passenden Kunden.

  erst nach (**insert** BESTELLUNG ...; **insert** KUNDE...).
- Einige explizite IBen oder IB-Kombinationen können gar nicht von einzelnen DB-Operationen erhalten werden, z. B.

  Zu jeder Bestellung gibt es einen passenden Kunden
  und jeder Kunde hat mindestens eine Bestellung.

  insert BESTELLUNG... nur zusammen mit **insert** KUNDE...!
- Deshalb sollten Transaktionen als kleinste Einheiten für persistente Änderungen und für die Erhaltung expliziter IBen dienen.

Transaktionen – konzeptionell und technisch

Def.: Eine **Transaktion** ist konzeptionell eine Folge von DB-Operationen,

- die die DB von einem zulässigen Zustand wieder in einen zulässigen Zustand überführt (bzgl. der statischen IBen),
- die einen zulässigen Zustandsübergang bewirkt (bzgl. der transitionalen IBen),
- und die nur als Einheit ("alles oder nichts") wirksam werden kann.

Man sagt auch, eine Transaktion muss das **ACID**-Prinzip erfüllen, also folgende Eigenschaften garantieren:

- Atomarität (**A**tomicity): s.o.
- Konsistenzerhaltung (**C**onsistency): s.o.
- Isolation (**I**solation): Gleichzeitig laufende Transaktionen beeinflussen sich nicht gegenseitig.
- Dauerhaftigkeit (**D**urability): Ergebnisse sind dauerhaft.

Transaktionen – konzeptionell und technisch (Forts.)

Kennzeichnung:	*in SQL:*
begin transaction	Sitzungs-/Programmbeginn oder letztes **commit** bzw. **rollback**
end transaction	(Freigabe aller Änderungen seit Beginn der Transaktion) **commit**, evtl. implizit, *z.B. in Oracle:* bei Sitzungsende oder DDL-Kommandos
undo transaction	(Rücknahme aller Änderungen seit Beginn der Transaktion) **rollback**

(Ideale) Formulierung von IBen in SQL

Grundform (in Anlehnung an SQL-Standard):

create assertion ⟨ IBName ⟩
[**immediate** | **deferred**] [2]
check (⟨ Bedingung ⟩)

immediate: sofort nach einer Änderungsoperation (voreingestellt)
deferred: erst am Ende einer Transaktion testen

⟨Bedingung⟩: wie in **where**-Klausel, jedoch geschlossene Bedingung an die gesamte Datenbank,
in der sogar mit **old**/**new** auf Tupel vor/nach der Operation bzw. Transaktion Bezug genommen werden darf

[2] vollständige Syntax dieser Zeile im SQL-Standard (voreingestellt: NOT DEFERRABLE INITIALLY IMMEDIATE):
<constraint characteristics> ::= <constraint check time> [[NOT] DEFERRABLE] | [NOT] DEFERRABLE [<constraint check time>]
<constraint check time> ::= INITIALLY DEFERRED | INITIALLY IMMEDIATE

(Ideale) Formulierung von IBen in SQL (Forts.)

Hier verwendete Abkürzungen:

- **create assertion** ... wird weggelassen.
- **check for all** ⟨Relationenliste⟩ (⟨Bedingung⟩) steht für [3] :

  check (**not exists**
  (**select** $*$ **from** ⟨ Relationenliste ⟩
  where ⟨ negierte Bedingung ⟩))

- (φ_1 **implies** φ_2) steht für (**not** φ_1 **or** φ_2)

In Oracle:

- nur einrelationale, tupelbezogene IBen
  als **check**-Klauseln in Column- oder Table-Constraints
- **immediate/deferred**-Option wählbar [4]

[3] Erinnere: $\forall(x\colon R)(\varphi) \Leftrightarrow \neg\exists(x\colon R)(\neg\varphi)$

[4] wie im Standard, für beliebige Column- oder Table-Constraints, also für **check**-Klauseln und sogar für inhärente IBen wie Fremdschlüssel

Beispiele für IBen

1. Kein Kundenkonto darf unter -1000 absinken.

   check for all KUNDE (Kto $\geq$ -1000)

2.1 Der Kunde 9876 darf sein Konto nicht überziehen.

   check for all KUNDE (KNr=9876 **implies** Kto $\geq$ 0)

2.2 Kunden aus Frankfurt dürfen ihre Konten nicht überziehen.

   . . . (KAdr **like** '%Frankfurt%' **implies** Kto $\geq$ 0)

*Bis hierher sind die IBen auch in Oracle ausdrückbar (***implies** *auflösen).*

---

3. Jede Ware muss von zwei Lieferanten angeboten werden.

```
deferred check for all ANGEBOT A1
  (exists (select * from ANGEBOT A2
   where A1.Ware=A2.Ware and A1.LName ≠ A2.LName))
```

Beispiele für IBen (Forts.)

4. Die Gesamtmenge aller Bestellungen einer Ware bei einem Lieferanten darf den zugehörigen Lagerbestand[5] nicht überschreiten.

```
deferred check for all ANGEBOT A
    A.Bestand >= (select sum(Menge) from BESTELLUNG
      where Ware = A.Ware and LName = A.LName)
```

5. Zu jeder Mehrfachbestellung muss ein Preis[5] bekannt sein.

```
deferred check for all BESTELLUNG B
  (Menge > 1 implies exists
    (select * from ANGEBOT
     where Ware = B.Ware and LName = B.LName
     and   Preis is not null))
```

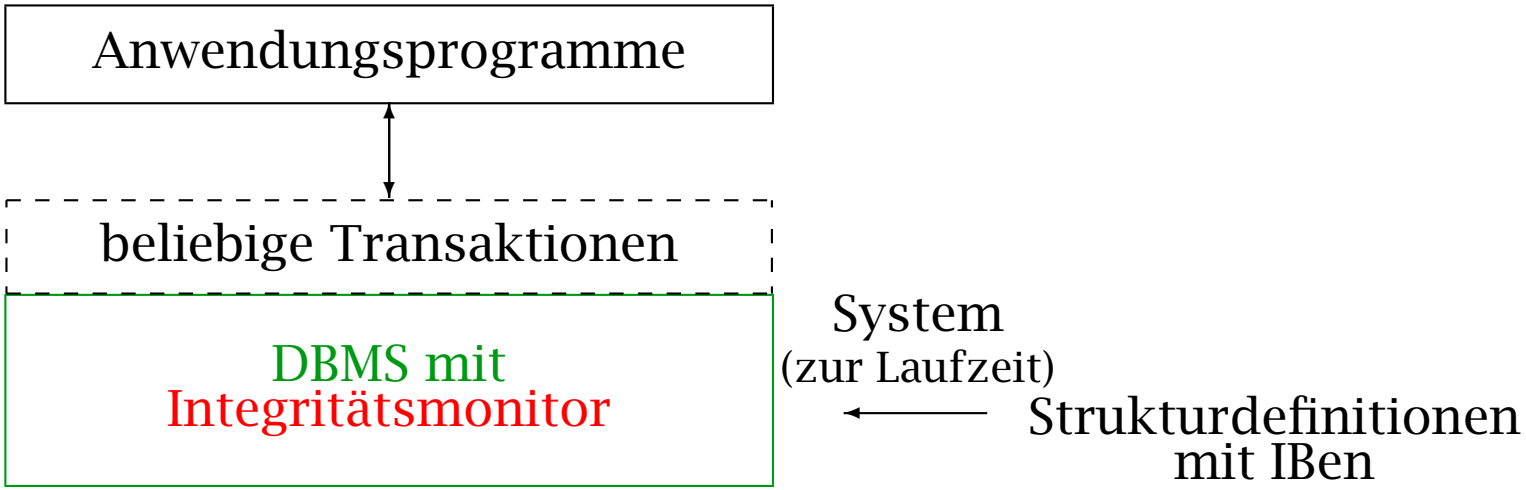
6. Bestellmengen für PCs dürfen nicht verringert werden.

```
check for all BESTELLUNG
  (Ware like 'PC%' implies new.Menge ≥ old.Menge)
```

[5] Der Lagerbestand sei in der Relation ANGEBOT als Attribut Bestand gespeichert. — Das Attribut Preis darf Nullwerte annehmen.

Alternativen der Integritätsüberwachung

(a) durch universellen (= anwendungsunabhängigen) Monitor:



zuverlässig, aber aufwändig, da jede IB

for all R x ... (ψ)

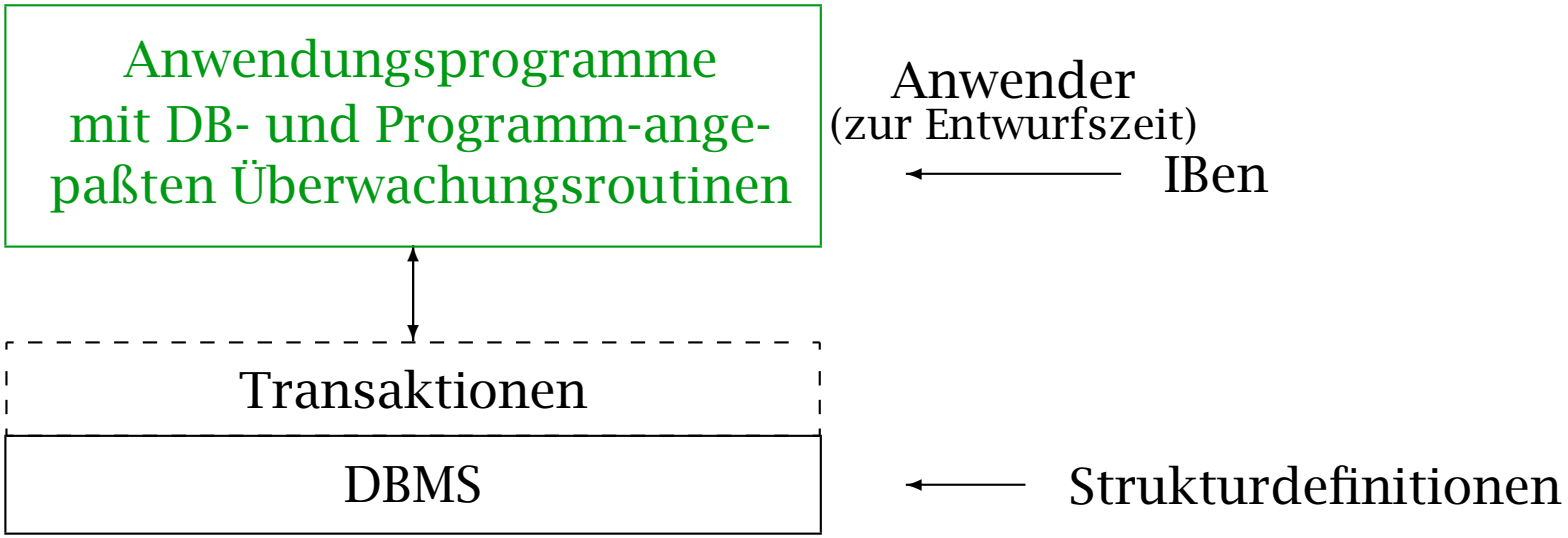
eine eigene Kontrollanfrage

not exists (**select** $*$ **from** R x ... **where not** ψ)

induziert

Alternativen der Integritätsüberwachung (Forts.)

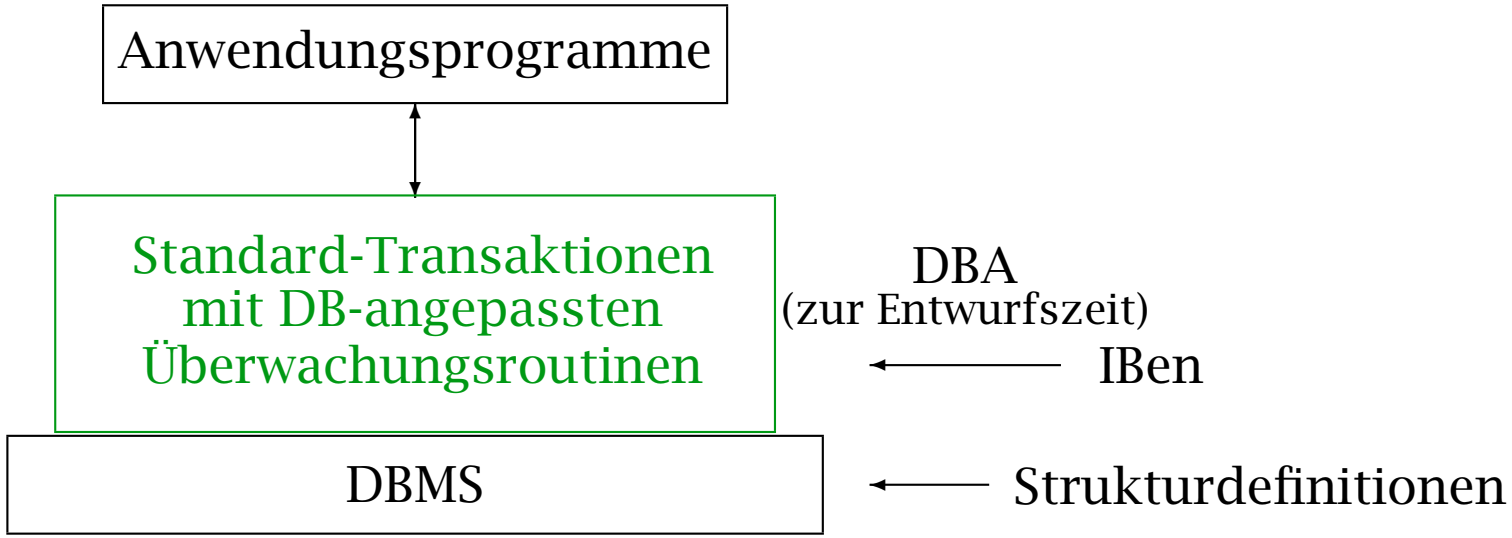
(b) lokal in Anwendungsprogrammen:



sehr effizient, aber ohne zentrale Verantwortung, oft unzuverlässig

Alternativen der Integritätsüberwachung (Forts.)

(c) zentral durch ausgewählte Transaktionen:



effizienter, unter zentraler Verantwortung, erlaubt aber keine Ad-hoc-Transaktionen mehr

Alternativen der Integritätsüberwachung (Forts.)

(d) mit Hilfe von Triggern: Aktivierung von DB-angepassten (also optimierten) Überwachungsroutinen nach beliebigen Transaktionen; quasi eine (bereits vom DB-Entwerfer bzw. -Administrator) optimierte Simulation eines Integritätsmonitors

Ein **Trigger** ist konzeptionell eine "Ereignis-Bedingungs-Aktions-Regel": (engl. *event-condition-action (ECA) – rule*)

Wenn

- zu spezifizierten Zeitpunkten (z. B. am Ende von Transaktionen oder zu Beginn von Änderungsoperationen)
- ein spezifiziertes Ereignis (z. B. eine **delete**-Operation auf einer benannten Relation) auftritt oder aufgetreten ist,
- und wenn eine spezifizierte Bedingung erfüllt ist,

dann

- wird eine spezifizierte Aktion ausgelöst.

6.2 Trigger in Oracle–PL/SQL

Ein **Trigger** in Oracle:

- ist ein PL/SQL-Block oder eine PL/SQL-Prozedur
  bezogen auf eine Tabelle, eine Sicht, ein Schema oder eine Datenbank
- wird implizit ausgeführt, wenn ein bestimmtes Ereignis eintritt

Auslösende Ereignisse können sein:

- Datenereignisse wie DML-Operationen auf Tabellen oder Sichten (z.B. **update**) oder DDL-Anweisungen (z.B. **alter**)
- Systemereignisse wie An- oder Abmelden von Schemata oder Datenbanken (z.B. login)

Hier betrachten wir nur DML-Trigger.

Arten von DML-Triggern

Der Typ des Triggers bestimmt, ob der Rumpf für jede Zeile oder nur einmal für die auslösende Anweisung ausgeführt wird.

- Ein **Anweisungs-Trigger (statement trigger):**
  - wird einmal für die auslösende Anweisung ausgeführt
  - ist der Default-Typ für Trigger
  - wird auch ausgeführt, wenn keine Zeile betroffen ist

- Ein **Zeilen-Trigger (row trigger):**
  - wird einmal für jede Zeile ausgeführt, die von der auslösenden Anweisung betroffen ist
  - ist zu erkennen an der **for-each-row**-Klausel
  - wird nicht ausgeführt, wenn das auslösende Ereignis keine einzige Zeile betrifft

Erzeugen von DML-Triggern

Erzeugen von Anweisungs- oder Zeilen-Triggern:

create [**or replace**] **trigger** *Triggername*
Timing Ereignis [**or** *Ereignis2* **or** *Ereignis3* ...] **on** *Objektname*
[[**referencing old as** *alt* | **new as** *neu*]
for each row
[**when** (*Bedingung*)]]
Trigger-Rumpf

Erzeugen von DML-Triggern (Forts.)

Zur obigen Syntax:

- *Triggername*: identifiziert den Trigger eindeutig (im User-Schema).
- *Timing*: gibt an, wann der Trigger relativ zum auslösenden Ereignis ausgeführt wird; erlaubte Werte sind **before**, **after** oder **instead of**.
- *Ereignis*: identifiziert die DML-Operation, die den Trigger auslöst; erlaubte Werte sind **insert**, **update** [**of** *Spalte(n)*] und **delete**.
- *Objektname*: gibt eine Tabelle oder Sicht an

Für Zeilen-Trigger kann man spezifizieren:

- **referencing**: gibt Aliasnamen *alt*/*neu* statt der Defaults **old**/**new** an, um sich auf die alten und neuen Werte der aktuellen Zeile zu beziehen (z.B. **old**.salary).
- **for each row**: gibt an, dass es sich um einen Zeilen-Trigger handelt.
- **when**: gibt eine *Bedingung* an, die für jede betroffene Zeile ausgewertet wird und so bestimmt, ob der Trigger-Rumpf für diese Zeile ausgeführt wird oder nicht

Der *Trigger-Rumpf* beinhaltet die Aktion, die vom Trigger ausgeführt werden soll, und wird wie folgt implementiert:

- als anonymer PL/SQL-Block mit **declare**/**begin**/**end**
- als **call** *Prozedurname(Parameter)*, um eine gespeicherte Prozedur zu starten

commit, **rollback** und **savepoint** sind darin nicht erlaubt.

Ausführung von Triggern

Wann soll ein Trigger ausgeführt werden?

- **before**: führe den Trigger-Rumpf "vor"[6] dem auslösenden DML-Ereignis auf der Tabelle/Sicht aus
- **after**: führe den Trigger-Rumpf nach dem auslösenden DML-Ereignis aus
- **instead of**: führe den Trigger-Rumpf anstelle der auslösenden DML-Operation aus. Diese Alternative wird für Sichten empfohlen, wenn das Update verboten ist oder eine unklare/unerwünschte Wirkung hat.

Ablaufrahmen aller Trigger zum selben auslösenden DML-Ereignis:

(a) Führe alle Anweisungs-Trigger mit einem **before** aus.

(b) Wiederhole für alle betroffenen Zeilen:

  (a) Führe alle Zeilen-Trigger mit einem **before** aus.

  (b) Führe die auslösende DML-Anweisung aus und prüfe die Integritätsbedingungen[7], die in der Tabellendefinition angegeben sind.

  (c) Führe alle Zeilen-Trigger mit einem **after** aus.

(c) Führe alle Anweisungs-Trigger mit einem **after** aus.

Sind mehrere Trigger für dasselbe Objekt definiert, ist ansonsten die Reihenfolge der Ausführung willkürlich.

---

[6] eigentlich: vor Beendigung des DML-Ereignisses, wenn die Werte vorher (old) noch und nachher (new) schon bekannt sind

[7] Diese Integritätsprüfung kann so lange verzögert werden (mit **deferred**), bis die **commit**-Operation ausgeführt ist. Oracle-Trigger selber sind leider nicht verzögerbar!

Ausführung von Triggern (Forts.)

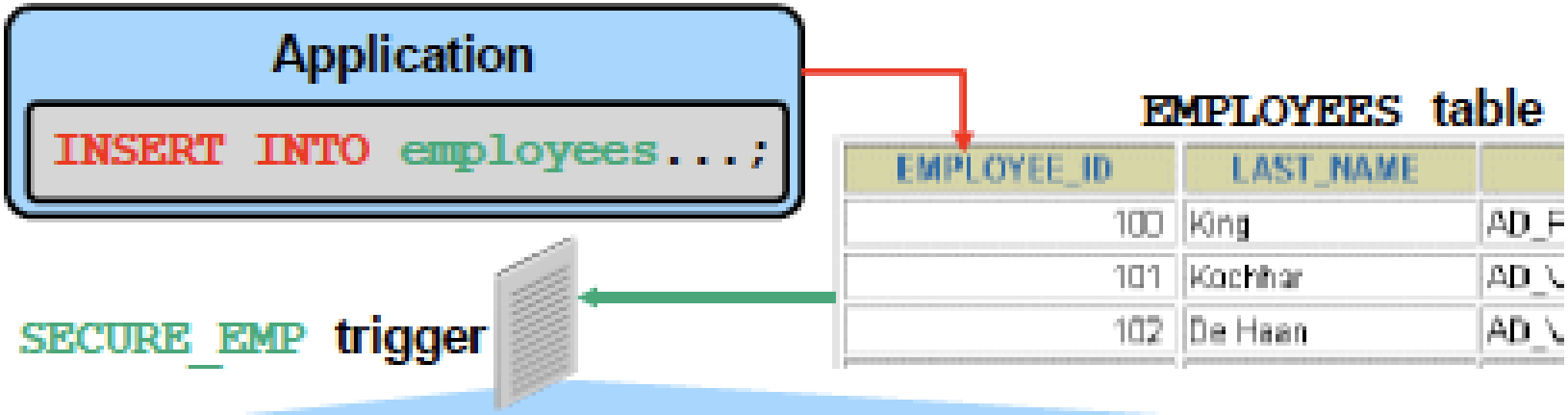
Ausführungssequenz aller Trigger zum selben auslösenden DML-Ereignis, z.B. zu **update of** salary **on** EMPLOYEES, wenn die Zeilen mit department_id = 30 betroffen sind wie etwa in folgender Anweisung:

```
update EMPLOYEES
set    salary = salary *1.1
where  department_id = 30;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

→ **before**-Anweisungs-Trigger
→ **before**-Zeilen-Trigger
→ **after**-Zeilen-Trigger
...
→ **before**-Zeilen-Trigger
→ **after**-Zeilen-Trigger
...
→ **after**-Anweisungs-Trigger

Beispiel: Erzeugen eines DML-Anweisungs-Triggers → *Datenschutz*




```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees BEGIN
 IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI')
           NOT BETWEEN '08:00' AND '18:00') THEN
  RAISE_APPLICATION_ERROR(-20500, 'You may insert'
    ||' into EMPLOYEES table only during '
    ||' business hours.');
  END IF;
END;
```

Beispiel: Testen des Triggers

Die folgende SQL-Anweisung

```
insert into EMPLOYEES (employee_id, last_name, first_name,
                       email, hire_date, job_id, salary, department_id)
values (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60);
```

werde am Samstag-Morgen abgesetzt. Reaktion:

```
→ insert into EMPLOYEES (employee_id, last_name, first_name,
          *
ERROR at line 1:
ORA-20500: You may insert into EMPLOYEES table only during business hours.
ORA-06512: at "PLSQL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'PLSQL.SECURE_EMP'
```

Verwenden bedingter Prädikate

→ *Datenschutz*

```
create or replace trigger secure_emp
before insert or update or delete on EMPLOYEES
begin
   if ((to_char(sysdate,'DY') IN ('SAT','SUN'))
       or (to_char(sysdate,'HH24') not between '08' and '18'))
      and user<>'DBA'
   then
      if deleting then raise_application_error(-20502,'You may delete '
                || 'from EMPLOYEES table only during business hours.');
      elsif inserting then raise_application_error(-20501,'You may insert '
                || 'into EMPLOYEES table only during business hours.');
      elsif updating('SALARY') then raise_application_error(-20503,'You may '
                || 'update salary only during business hours.');
      else raise_application_error(-20504,'Normal users may '
                || 'update EMPLOYEES table only during normal hours.');
      end if;
   end if;
end;
```

Beispiel: Erzeugen eines DML-Zeilen-Triggers → *Integritätssicherung*

```
create or replace trigger restrict_salary
before insert or update of salary, job_id ON EMPLOYEES
for each row
begin
  if not (:new.job_id IN ('AD_PRES','AD_VP') or :new.salary <= 15000)
   then raise_application_error (-20202,
      'Employee cannot earn more than $15,000.');
   end if;
end;
```

Bemerkung zu diesem konkreten Beispiel: Dieser Trigger überwacht die Integritätsbedingung: Jeder Angestellte, der nicht Präsident oder Vizepräsident ist, verdient höchstens 15000.

Eigentlich wird kein Trigger benötigt, um solch eine einfache Integritätsbedingung durchzusetzen (die sich nur auf jede Zeile einzeln bezieht); es könnte dafür ein Oracle-**check**-Constraint angegeben werden.

Zweites Beispiel: Erzeugen eines DML-Zeilen-Triggers → *Integr.sichg.*

```
create or replace trigger check_manager
before insert or update of manager_id on EMPLOYEES
for each row
when (new.manager_id is not null)
declare dmgrid DEPARTMENTS.manager_id%TYPE;
begin
   select manager_id into dmgrid
   from DEPARTMENTS where department_id=:new.department_id;
   if not :new.manager_id = dmgrid
   then raise_application_error (-20203,
    'Inconsistent manager information in EMPLOYEES/DEPARTMENTS');
   end if;
end;
```

Bemerkung: Hier geht es um die Integritätsbedingung: Der Manager (falls angegeben) eines Angestellten ist der Manager der Abteilung des Angestellten.

Diese kann nicht durch ein Oracle-**check**-Constraint formuliert werden.

Achtung: Kein Doppelpunkt vor **new**/**old** in **when**-Klausel! Keine Unteranfragen!

Zweites Beispiel: Erzeugen eines DML-Zeilen-Triggers (Forts.)

→ *Integritätssicherung*

Aktive Variante:

```
create or replace trigger correct_manager
before insert or update of manager_id on EMPLOYEES
for each row
when (new.manager_id is not null)
declare dmgrid DEPARTMENTS.manager_id%TYPE;
begin
   select manager_id into dmgrid
   from DEPARTMENTS where department_id=:new.department_id;
   :new.manager_id := dmgrid;
   DBMS_OUTPUT.PUT_LINE('Manager input ignored');
end;
```

Drittes Beispiel: Erzeugen eines DML-Zeilen-Triggers

→ *Berechnung abgeleiteter Daten*

```
create or replace trigger derive_commission_pct
before insert or update of salary on EMPLOYEES
for each row
when (new.job_id = 'SA_REP')
begin
   if inserting then
      :new.commission_pct := 0;
   elsif :old.commission_pct is null then
      :new.commission_pct := 0;
   else
      :new.commission_pct:= :old.commission_pct+0.05;
   end if;
end;
```

Viertes Beispiel: Erzeugen eines DML-Zeilen-Triggers → *Auditing*

```
create or replace trigger AUDIT_EMP_values
after delete or insert or update on EMPLOYEES
for each row
begin
   insert into AUDIT_EMP(user_name, time_stamp, id,
      old_last_name, new_last_name, old_title, new_title,
      old_salary, new_salary)
   values (user, sysdate, :old.employee_id,
      :old.last_name, :new.last_name, :old.job_id, :new.job_id,
      :old.salary, :new.salary);
end;
```

Viertes Beispiel: Erzeugen eines DML-Zeilen-Triggers (Forts.)

Auswirkungen des Auditing-Triggers:
(beachte Nullwerte für **new**/**old**-Felder)

```
insert into EMPLOYEES
            (employee_id, last_name, job_id, salary, ...)
values      (999, 'Temp emp', 'SA_REP', 1000,...);

update EMPLOYEES
set         salary = 2000, last_name = 'Smith'
where       employee_id = 999;

delete  from EMPLOYEES
where       employee_id = 999;

select  *  from AUDIT_EMP;          -- Zwischenstand der Audit-Tabelle
```

USER_NAME	TIMESTAMP	ID	OLD_LAST_N	NEW_LAST_N	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
PLSQL	28-SEP-01			Temp emp		SA_REP		1000
PLSQL	28-SEP-01	999	Temp emp	Smith	SA_REP	SA_REP	1000	2000
PLSQL	28-SEP-01	999	Smith		SA_REP		2000	

Das (lästige) Problem der mutierenden Tabellen

```
create or replace trigger check_salary
before insert or update of salary, job_id on EMPLOYEES
for each row
when (new.job_id <> 'AD_PRES')
declare
   minsalary EMPLOYEES.salary%TYPE;
   maxsalary EMPLOYEES.salary%TYPE;
begin
   select min(salary), max(salary) into minsalary, maxsalary
   from EMPLOYEES
   where job_id = :new.job_id;
   if :new.salary < minsalary OR :new.salary > maxsalary then
      raise_application_error(-20505,'Salary out of range');
   end if;
end;
```

Das (lästige) Problem der mutierenden Tabellen (Forts.)

```
update EMPLOYEES
set    salary = 3400
where  last_name = 'Stiles';
```

```
→ update EMPLOYEES
          *
  ERROR at line 1
  ORA-04091: table PLSQL.EMPLOYEES is mutating, trigger/function
  may not see it
  ORA-06512: at 'PLSQL.CHECK_SALARY', line 5
  ORA-04088: error during execution of trigger 'PLSQL.CHECK_SALARY'
```

Diese Restriktion hindert den Zeilen-Trigger an der Erkennung einer inkonsistenten Datenmenge.

Ein möglicher Lösungsansatz für das Problem der mutierenden Tabellen:

- Speichere die benötigten Daten (hier die kleinsten und die größten Gehälter) in einer anderen Tabelle, die mit Hilfe anderer (z.B. Anweisungs-) DML-Trigger aktuell gehalten wird.

Management von Triggern

- Deaktivieren oder Reaktivieren eines Datenbank-Triggers:

  alter trigger *trigger_name* **disable | enable**

- Deaktivieren oder Reaktivieren aller Trigger einer Tabelle:

  alter table *table_name* **disable | enable all triggers**

- Erneutes Kompilieren eines Triggers einer Tabelle:

  alter trigger *trigger_name* **compile**

- Entfernen von Triggern aus der Datenbank:

  drop trigger *trigger_name*

- Beachte: Alle Trigger auf einer Tabelle werden entfernt, wenn die Tabelle gelöscht wird.

Informationen über Trigger im Data-Dictionary

- USER_OBJECTS: Objektinformationen, auch zu Triggern
- USER_TRIGGERS: strukturierte Triggertexte

Spalte	Spaltenbeschreibung
TRIGGER_NAME	Name des Triggers
TRIGGER_TYPE	Timing des Triggers: **before**|**after**|**instead of**
TRIGGERING_EVENT	auslösende DML-Anweisungen
TABLE_NAME	Name der Datenbanktabelle/-sicht
REFERENCING_NAMES	Aliase für **:old** und **:new**
WHEN_CLAUSE	**when**-Klausel
TRIGGER_BODY	Trigger-Rumpf (PL/SQL-Aktion)
STATUS	Status des Triggers: enabled|disabled

- nicht in USER_SOURCE
- USER_ERRORS: Kompilierungsfehler, auch zu Triggern

Informationen über Trigger im Data-Dictionary (Forts.)

Beispiel:

```
select  trigger_name, trigger_type, triggering_event, table_name,
        referencing_names, status, trigger_body
from    USER_TRIGGERS
where trigger_name = 'RESTRICT_SALARY';
```

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	REFERENCING_NAMES	WHEN_CLAUS	STATUS	TRIGGER_BODY
RESTRICT_SALARY	BEFORE EACH ROW	INSERT OR UPDATE	EMPLOYEES	REFERENCING NEW AS NEW OLD AS OLD		ENABLED	BEGIN IF NOT (:NEW.JOB_ ID IN ('AD_PRES ', 'AD_VP')) AND :NE W.SAL

...

6.3 Integritätsüberwachung durch Trigger

Grundmuster der Monitorsimulation:

create trigger monitor_⟨IB-Name⟩
deferred after ⟨DB-Operationen⟩ (Ereignisse)
when (⟨negierte IB⟩) (Bedingung)
rollback (Aktion)

Diese Lösung wäre allerdings genauso aufwändig wie der Monitor selbst.

Da induktiv *angenommen* werden kann, dass die IB jeweils im *Vor*-Zustand einer Transaktion erfüllt war, lässt sich der Trigger "spezialisieren" (nämlich auf "kritische" DB-Operationen und -Objekte), um die IB im *Nach*-Zustand zu garantieren.

Somit sind *beim DB-Entwurf*

- nicht nur alle IBen zu spezifizieren,
- sondern zu diesen IBen auch zugehörige spezialisierte Trigger zu programmieren.

Integritätsüberwachung durch Trigger (Forts.)

Optimierte Monitorsimulation:

create trigger monitor_⟨IB-Name⟩
deferred after ⟨kritische DB-Operationen⟩
when (⟨spezialisierte negierte IB⟩)
⟨Reaktion:⟩
- passive Integritätsüberwachung: **rollback**
- aktive Integritätssicherung: ⟨Korrektur auf kritischen DB-Objekten⟩

- kritische DB-Operationen:
  Operationen, die IB-Verletzungen bewirken können
- spezialisierte IB: auf kritische DB-Operationen/Objekte zugeschnitten
- kritische DB-Objekte:
  Daten, die an IB-Verletzungen beteiligt sein können, und die von kritischen DB-Operationen (evtl. indirekt) betroffen sind
- Korrektur: Folge von geeigneten DB-Operationen

Kritische DB-Operationen

grobes Schema: Welche DB-Operationen sind für welche IB "kritisch" ?

Operation: / *IB in idealem SQL:*	**insert** R	**delete** R	**update** R(A)	**update** R(B) ($B \not\equiv A$)
for all R r ... r.A ...	×	—	×	—
(nicht negiertes) **exists** ... ...**from** R r ... r.A ...	—	×	×	—
R nicht erwähnt	—	—	—	—
Aggregierung (z.B. sum) über R.A	×	×	×	—

Beachte: Dieses Schema klassifiziert u. U. zu viele Operationen als "kritisch". — Generell brauchen bei der Analyse auf kritische Operationen solche Verletzungen von IBen, die auch die Verletzung einer inhärenten IB wie z.B. einer Fremdschlüsselbedingung implizieren, nicht betrachtet werden. Denn für inhärente IBen kann man sich auf eine immediate-Sicherung verlassen. Somit kommen für manche IBen weniger Operationen als Verursacher IB-spezifischer Verletzungen in Frage.

Integritätsüberwachung mit Oracle-Triggern

Vgl. Beispiele im Abschnitt 6.2

Beobachtungen:

+ Zeilen-Trigger (“**for each row**”) bieten eine Spezialisierung auf die Tupel, die von auslösenden DB-Operationen (direkt) betroffen sind.
– “mutating table”-Einschränkung bei Zeilen-Triggern
+ **old**/**new**–Bezüge helfen, sogar transitionale IBen zu prüfen.
+ **before**-Trigger helfen, die Triggerausführung zu optimieren
– Anweisungs-Trigger erlauben keine **old**/**new**–Bezüge und keine operationsabhängige Spezialisierung.
– nur **immediate**-Ausführung, keine **deferred**-Trigger

Erfordernisse von idealen deferred-Triggern *(nicht in Oracle)*

deferred-Trigger sind nur in Kombination mit **after** sinnvoll.

Um eine Spezialisierung auf betroffene Tupel zu erlauben, müssen die während einer Transaktion geänderten Tupel in sogenannten **Differenz-** oder **Transitionstabellen** zu jeder Relation R gespeichert werden[8]:

- deleted_R: aus R gelöschte Tupel
- updated_old_R: in R upgedatete Tupel vor der Transaktion
- updated_new_R: in R upgedatete Tupel nach der Transaktion
- inserted_R: in R eingefügte Tupel

  (Evtl. sind deleted/updated_old bzw. inserted/updated_new zusammengefasst.)

Dann können sich Bedingungs- und Aktionsteil darauf beziehen, auch in Anweisungs-Triggern. Vor- und Nachzustand der Transaktion lassen sich mittels Differenztabellen ineinander umrechnen[9].

Solche Differenztabellen lassen sich übrigens durch **immediate**-Zeilentrigger füllen *(wie?)*, wären also auch mit Oracle erzeugbar.

---

[8] Immerhin sieht der SQL-Standard solche Transitionstabellen old/new R nach je einem SQL-Statement vor.

[9] Die Tupel werden mit ihrer Originaladresse (Rowid) gespeichert. Damit lassen sich Tupel, die in updated_old/_new zusammengehören , identifizieren. Es gilt: **new** R = **old** R − deleted_R − updated_old_R ∪ updated_new_R ∪ inserted_R. Man beachte, dass nur der "Nettoeffekt" der Transaktion dargestellt wird; z.B. hat die Folge $\mathsf{insert}(t); \mathsf{update}(t, t')$ den gleichen Effekt wie $\mathsf{insert}(t')$.

Beispiele für (ideale) deferred-Anweisungs-Trigger

Beispiel-IB: **deferred check forall** BESTELLUNG B
exists (**select** ∗ **from** KUNDE K **where** K.KNr = B.KNr **and** K.Kto>99)

kritische Operationen:
delete KUNDE, **insert** BESTELLUNG, **update** KUNDE.Kto[10]

Zugehörige optimierte Trigger lauten u.a.:

- **create trigger** restrict_ins_BEST
  deferred after insert on BESTELLUNG
  when exists (**select** ∗ **from** inserted_BESTELLUNG B
  where not exists (**select** ∗ **from** KUNDE K
  where K.KNr=B.KNr **and** K.Kto>99))
  begin print("unpassende(r) Besteller"); **rollback end**

- **create trigger** cascade_upd_KUNDE
  deferred after update of Kto **on** KUNDE
  delete from BESTELLUNG B
  where exists (**select** ∗ **from** updated_new_KUNDE K
  where K.KNr=B.KNr **and** K.Kto<=99)

[10] **update**s des Attributs KNr seien hier ausgeschlossen, da KNr Teil des Primärschlüssels.

Beispiele für (ideale) deferred-Anweisungs-Trigger (Forts.)

Weiteres Beispiel für die Entwicklung spezialisierter Trigger:

Schemata:

EMP[LOYEE] (Name,. . . ,DeptId),
DEP[ARTMEN]T (Id,. . .)

IB:
- **deferred check forall** DEPT D
  (**select count**(∗) **from** EMP **where** DeptId = D.Id) ≤ 100

Um nicht nach jeder Transaktion auf EMP neu zählen zu müssen, erweitere DEPT *um ein (verstecktes) redundantes Attribut* Counter.

Damit sind folgende Ersatz-IBen zu überwachen:

- **deferred check forall** DEPT (Counter ≤ 100)
- **deferred check forall** DEPT D (D.Counter =
  select count(∗) **from** EMP **where** DeptId = D.Id)

Beispiele für (ideale) deferred-Anweisungs-Trigger (Forts.)

Trigger:

create trigger maintain_DEPT_Counter
deferred after insert or delete or update of DeptId **on** EMP:
  update DEPT D **set** D.Counter = D.Counter
  − (**select count**(∗) **from** deleted_EMP **where** DeptId = D.Id)
  − . . . updated_old_EMP . . .
  + . . . inserted_EMP . . .
  + . . . updated_new_EMP . . .

(dadurch ausgelöster) Folgetrigger:

create trigger monitor_DEPT_Counter
immediate after update of Counter **on** DEPT:
when exists (**select** ∗ **from** updated_new_DEPT
  where Counter > 100)
  rollback

Bemerkungen zu Triggern

Beobachtungen:

- Nach einer Transaktion können mehrere Trigger auslösbar sein.
- Trigger können andere Trigger auslösen.

⇒ *Probleme:*

- Eindeutigkeit des Ergebnisses nach Abarbeitung aller Trigger ?
- Terminierung ?
- Wartbarkeit komplexer Triggersysteme ?

Andere Anwendungen von Triggern (neben Integritätssicherung):

- Datenschutz[11]
- Auditing, Logging
- Sichten-Updates[12]
- Berechnung redundanter/abgeleiteter Daten
- Wartung materialisierter Sichten
- Replikation in verteilten Datenbanken

[11] z.B. Kontrollen, die vom Benutzernamen und von der aktuellen Zeit abhängig sind (Pseudoattribute `user`, `sysdate`)
[12] Programmierung von sichtenspezifischen Updateprogrammen mittels **instead of**- Triggern