
Kapitel 3

Datenintegrität

Folien zum Datenbankpraktikum
Wintersemester 2009/10 LMU München

© 2008 Thomas Bernecker, Tobias Emrich
unter Verwendung der Folien des Datenbankpraktikums aus dem Wintersemester 2007/08 von Dr. Matthias Schubert

Übersicht

3.1 Integritätsbedingungen

3.2 Deklarative Constraints

3.3 Prozedurale Constraints (Trigger)

Integritätsbedingungen (Integrity Constraints)

- Bedingungen, die von einer Datenbank zu jedem Zeitpunkt erfüllt sein müssen
- Einschränkungen der möglichen **DB-Zustände** (Ausprägungen der Relationen)
- Einschränkungen der möglichen **Zustandsübergänge** (Update-Operationen)
- Von wem werden Integritätsbedingungen überwacht:
 - vom DBMS?
 - vom Anwendungsprogramm?

- Integritätsbedingungen sind Teil des Datenmodells
 - wünschenswert ist eine zentrale Überwachung im DBMS innerhalb des Transaktionsmanagements
 - Einhaltung unabhängig von der jeweiligen Anwendung gewährleistet, es gelten dieselben Integritätsbedingungen für alle Benutzer
- für eine Teilmenge von Integritätsbedingungen (`primary key`, `unique`, `foreign key`, `not null`, `check`) ist dies bei den meisten relationalen Datenbanken realisiert
- für eine allgemeinere Klasse von Integritätsbedingungen häufig Definition und Realisierung im Anwendungsprogramm notwendig
 - Problem: Nur bei Verwendung des jeweiligen Anwendungsprogrammes ist die Einhaltung der Integritätsbedingungen garantiert, Korrektheit, ...
- ORACLE: einfache Integritätsbedingungen direkt in DDL (**deklarativ**), Unterstützung für komplexere Integritätsbedingungen durch Trigger-Mechanismus (**prozedural**)

Übersicht

3.1 Integritätsbedingungen

3.2 Deklarative Constraints

3.3 Prozedurale Constraints (Trigger)

Deklarative Constraints

- Teil der Schemadefinition (`create table ...`)
- Arten:
 - Schlüsseleigenschaft: `primary key` (einmal), `unique` (beliebig)
 - keine Nullwerte: `not null` (implizit bei `primary key`)
 - Typintegrität: `Datentyp`
 - Wertebedingungen: `check (<Bedingung>)`
 - referentielle Integrität: `foreign key ... references ...` (nur Schlüssel)
- Constraints können **attributsbezogen** (für jeweils ein Attribut) und **tabellenbezogen** (für mehrere Attribute) definiert werden.
- Beschreibung möglich durch geschlossene logische Formeln (Sätze) 1.Stufe.
Bsp. (Notation wie Tupelkalkül): Es darf keine zwei Räume mit gleicher R_ID geben.
$$IB_1 : \forall r_1 \in \text{Raum} (\forall r_2 \in \text{Raum} (r_1[R_ID] = r_2[R_ID] \Rightarrow r_1 = r_2))$$

Für jede Belegung muss ein entsprechender Raum existieren.
$$IB_2 : \forall b \in \text{Belegung} (\exists r \in \text{Raum} (b[R_ID] = r[R_ID]))$$

○ Definition des Beispiels in SQL:

- Bei *IB1* handelt es sich um eine Eindeutigkeitsanforderung an die Attributswerte von *R_ID* in der Relation *Raum* (Schlüsseleigenschaft).
- *IB2* fordert die referentielle Integrität der Attributswerte von *R_ID* in der Relation *Belegung* als Fremdschlüssel aus der Relation *Raum*.

```
CREATE TABLE raum
(
    r_id  varchar2(10)          UNIQUE,                (IB1)
    ...                          PRIMARY KEY
)
```

```
CREATE TABLE belegung (
    b_id  number(10),
    r_id  varchar2(10)          CONSTRAINT fk_belegung_raum    (IB2)
                                     REFERENCES raum(r_id)
                                     raum
                                     [ON DELETE CASCADE],
    ...
)
```

○ Weiteres Beispiel:

```
CREATE TABLE stadt
(
    name varchar2(25) NOT NULL,                attributsbezogen
    bezirk varchar2(25) NOT NULL,              attributsbezogen
    kfz_kennz char(4),
    flaeche number(10,2) NOT NULL              attributsbezogen
    CONSTRAINT check_flaeche CHECK (flaeche > 0), attributsbezogen
    CONSTRAINT pk PRIMARY KEY (name,bezirk)    tabellenbezogen
)
```

```
ALTER TABLE stadt DISABLE CONSTRAINT check_flaeche;
```

Überwachung von Integritätsbedingungen durch das DBMS

Definitionen:

- **S** sei ein Datenbankschema
- **IB** sei eine Menge von Integritätsbedingungen **I** über dem Schema **S**
- **DB** sei Instanz von **S**, d.h. der aktuelle Datenbankzustand (über dem Schema **S**)
- **U** sei eine Update-Transaktion, d.h. eine Menge zusammengehöriger Einfüge-, Lösch- und Änderungsoperationen
- **U(DB)** sei der aktuelle Datenbankzustand nach Ausführen von **U** auf **DB**
- **Check (I, DB)** bezeichne den Test der Integritätsbedingung **I** auf dem aktuellen Datenbankzustand **DB**

$$\text{Check (I, DB)} = \begin{cases} \text{true, falls I in DB erfüllt ist} \\ \text{false, falls I in DB nicht erfüllt ist} \end{cases}$$

Prüfen der Integrität

Wann sollen Integritätsbedingungen geprüft werden?

- Periodisches Prüfen der Datenbank **DB** gegen alle Integritätsbedingungen:

```
if ((Check (I, DB) = true)) then <ok>
else <Rücksetzen auf letzten konsistenten Zustand>;
```

Probleme:

- Rücksetzen auf letzten geprüften konsistenten Zustand ist aufwändig
- beim Rücksetzen gehen auch korrekte Updates verloren
- erfolgte lesende Zugriffe auf inkonsistente Daten sind nicht mehr rückgängig zu machen

- Inkrementelle Überprüfung bei jedem Update
 - Voraussetzung: Update erfolgt auf einem konsistenten Datenbankzustand
 - dazu folgende Erweiterung:

$$\text{Check}(I, U, DB) = \begin{cases} \text{true, falls } I \text{ durch Update } U \text{ auf } DB \text{ nicht verletzt ist} \\ \text{false, falls } I \text{ durch Update } U \text{ auf } DB \text{ verletzt ist} \end{cases}$$

dann:

```
<führe U durch>;
```

```
if ( $\forall I \in IB$  (Check (I, U, DB) = true)) then <ok>  
  else <rollback U>;
```

- bei jedem Update **U** alle Integritätsbedingungen gegen die gesamte Datenbank zu testen ist zu teuer, daher Verbesserungen:
 1. nur betroffene Integritätsbedingungen testen; z.B. kann die referentielle Integritätsbedingung *Belegung* \rightarrow *Raum*, nicht durch
 - Änderungen an der Relation *Dozent* verletzt werden
 - Einfügen in die Relation *Raum* verletzt werden
 - Löschen aus der Relation *Belegung* verletzt werden
 2. abhängig von **U** nur vereinfachte Form der betroffenen Integritätsbedingungen testen; z.B. muss bei Einfügen einer *Belegung* **x** nicht die gesamte Bedingung **IB₂** getestet werden, sondern es reicht der Test von:

$$\exists r \in \text{Raum } (x[R_ID] = r[R_ID])$$

Bei welchen Operationen muss geprüft werden?

Beispiel:

```
create table ta
(a_id ...
...
...
primary key (a_id))

create table tb
(b_id ...
a_id
...
primary key (b_id),
foreign key (a_id) references ta)
```

- Insert/Update in Tabelle **tb**:

Existiert zu einem eingefügten/geänderten Fremdschlüssel in **tb** kein entsprechender Schlüssel in **ta**, dann wird die Operation zurückgewiesen.

- Update in Tabelle **ta**:

Existiert zu einem Schlüssel in **ta** ein abhängiger Datensatz (Fremdschlüssel) in **tb**, dann wird jede Änderung des Schlüssels zurückgewiesen.

- Löschen in Tabelle **ta**:

- Löschen immer möglich, wenn kein abhängiger Datensatz in **tb** existiert
- weitere Optionen:

Option	Wirkung
ON DELETE NO ACTION	Verifikation durch Trigger (default)
ON DELETE RESTRICT	Änderungsoperation wird zurückgewiesen
ON DELETE CASCADE	Abhängige Datensätze in tb werden automatisch gelöscht; kann sich über mehrstufige Abhängigkeiten fortsetzen
ON DELETE SET NULL	Wert des abhängigen Fremdschlüssels tb wird auf null gesetzt
ON DELETE SET DEFAULT	Wert des abhängigen Fremdschlüssels tb wird auf den Default-Wert der Spalte gesetzt

- Beispiel: `foreign key (a_id) references ta on delete cascade`

Übersicht

3.1 Integritätsbedingungen

3.2 Deklarative Constraints

3.3 Prozedurale Constraints (Trigger)

Prozedurale Constraints (Trigger)

- Motivation: Komplexere Bedingungen als bei deklarativen Constraints und damit verbundene Aktionen wünschenswert.
→ **Trigger**: Aktion (PL/SQL-Programm), die einer Tabelle zugeordnet ist und durch ein bestimmtes Ereignis ausgelöst wird.
- Ein Trigger enthält Code, der die mögliche Verletzung einer Integritätsbedingung bei einem bestimmten Ereignis-Typ testet und daraufhin bestimmte Aktionen veranlasst.
- mögliche Ereignisse: `insert`, `update`, `delete`
- zwei Arten:
 - **Befehls-Trigger** (*statement trigger*): werden einmal pro auslösendem Befehl ausgeführt.
 - **Datensatz-Trigger** (*row trigger*): werden einmal pro geändertem/eingefügtem/gelöschem Datensatz ausgeführt.
- mögliche Zeitpunkte: vor (BEFORE) oder nach (AFTER) dem auslösenden Befehl

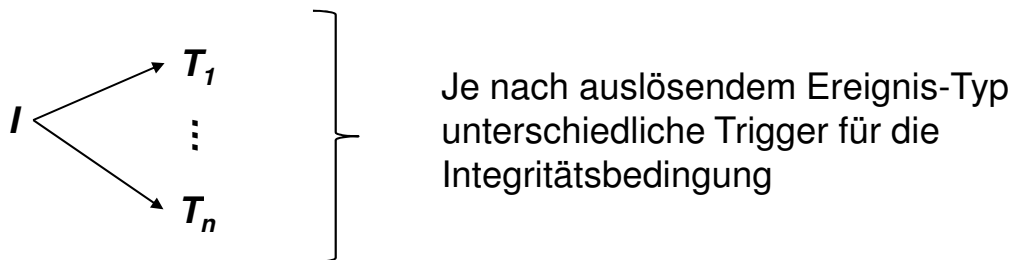
- Datensatz-Trigger haben Zugriff auf zwei Instanzen eines Datensatzes: vor und nach dem Ereignis (Einfügen/Ändern/Löschen)

Adressierung durch Präfix:

`:new.` bzw. `:old.` (PL/SQL-Block)

`new.` bzw. `old.` (Trigger-Restriktion)

- Zu einer Integritätsbedingung I gehören in der Regel mehrere Trigger T_i



- Aufbau eines Trigger-Programms:

```
create or replace trigger <trig_name>
before/after/instead of  -- Trigger vor/nach/alt. zu Auslöser ausführen
insert or                -- Trigger-Ereignisse
update of <attrib1>, <attrib2>, ... or
delete on <tab_name>/<view_name>/ -- zugehörige Tabelle od. View (DML-Trigger)
         <schema_name>/<db_name>  -- Schema od. Datenbank (DDL-Trigger)
[for each row]           -- Datensatz-Trigger
when <bedingung>         -- zusätzliche Trigger-Restriktion
declare                  -- PL/SQL-Block
...
begin
if inserting then <pl/sql>
if updating (<attrib1>) then <pl/sql>
if deleting then <pl/sql>
...
end;
```

Beispiel

- Ausgangspunkt: Relation *Period_Belegung* mit regelmäßig stattfindenden Lehrveranstaltungen in einem Hörsaal
- Hier müssen folgende Bedingungen gelten:

$$\forall p \in \text{Period_Belegung} \ (0 \leq p[\text{Tag}] \leq 6 \wedge p[E_Termin] \leq p[L_Termin] \\ \wedge p[\text{Tag}] = \text{day}(p[E_Termin]) \wedge p[\text{Tag}] = \text{day}(p[L_Termin]))$$

→ Formulierung als deklaratives Constraint:

```
SQL> ALTER TABLE Period_Belegung ADD CONSTRAINT check_day
CHECK (
    (Tag between 0 and 6) and
    (Erster_Termin <= Letzter_Termin) and
    (to_number (to_char (Erster_Termin, 'd')) = Tag) and
    (to_number (to_char (Letzter_Termin, 'd')) = Tag)
);
```

Formulierung als prozedurales Constraint (Trigger):

```
SQL> CREATE OR REPLACE TRIGGER check_day
BEFORE
INSERT OR UPDATE
ON Period_Belegung
FOR EACH ROW
DECLARE
    tag number; et date; lt date;
BEGIN
    tag := :new.Tag;
    et := :new.Erster_Termin;
    lt := :new.Letzter_Termin;
    if (tag < 0) or (tag > 6) or (et > lt) or
        (day(et) != tag) or (day(lt) != tag) then
        raise_application_error (-20089, 'Falsche Tagesangabe');
    end if;
END;
```

Sequenzen

- für die Erstellung eindeutiger IDs
- Beispiel:

```
CREATE SEQUENCE <seq_name>
[INCREMENT BY n]                -- Default: 1
[START WITH n]                  -- Default: 1
[{MAXVALUE n | NOMAXVALUE}]     -- Maximalwert (n | 10^27)
[{MINVALUE n | NOMINVALUE}]     -- Mindestwert (n | -10^26)
[{CYCLE | NOCYCLE}]             -- zyklisch oder nicht
[{CACHE n | NOCACHE}];         -- Vorcachen von Nummern
```

- Zugreifen über **NEXTVAL** (nächster Wert) und **CURRVAL** (aktueller Wert):

```
CREATE SEQUENCE seq_pers;
INSERT INTO person (p_id, name, alter)
VALUES (seq_pers.NEXTVAL, 'Ulf Mustermann', 28);
```

- Beispiel mit Trigger:

```
CREATE OR REPLACE TRIGGER pers_insert
BEFORE INSERT FOR EACH ROW
BEGIN
    SELECT seq_pers.NEXTVAL
    INTO :new.p_id
    FROM dual;
END;

INSERT INTO person (name, alter)
VALUES ('Ulf Mustermann', 28);
```

→ kein expliziter Zugriff (NEXTVAL) nötig!