LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Skript zur Vorlesung:

Datenbanksysteme I

Wintersemester 2016/2017

Kapitel 10
Transaktionen – Integrität

Vorlesung: : Prof. Dr. Christian Böhm
Übungen: Dominik Mautz

http://www.dbs.ifi.lmu.de/Lehre/DBS

Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Synchronisation** (Concurrency Control)
   Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer
2. **Datensicherheit** (Recovery)
   Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
3. **Integrität** (Integrity)
   Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer

Integritätsbedingungen

- Integritätsbedingungen (Integrity Constraints)
  - Bedingungen, die von einer Datenbank zu jedem Zeitpunkt erfüllt sein müssen
  - Typen
    - Schlüssel-Integrität
    - Referentielle Integrität
    - Multiplizitäten Constraints
    - Allgemeine Constraints
  - Diese Constraints sind
    - statisch, d.h. sie definieren Einschränkungen der möglichen ***DB-Zustände*** (Ausprägungen der Relationen)
    - dynamisch, d.h. sie spezifizieren Einschränkungen der möglichen **Zustandsübergänge** (Update-Operationen)

Integritätsbedingungen

- Beispiele
  - Eindeutigkeit von Schlüssel-Attributen (Schlüssel-Integrität)
  - Ein Fremdschlüssel, der in einer anderen Relation auf seine Basisrelation verweist, muss in dieser Basisrelation tatsächlich existieren (Referentielle Integrität)
  - Bei 1:*m*-Beziehungen müssen die Kardinalitäten beachtet werden – funktioniert z.B. durch Umsetzung mittels Fremdschlüssel auf der *m*-Seite (Multiplizitäten Constraint)
  - Wertebereiche für Attribute müssen eingehalten werden (allgemeines Constraint)

    Achtung: das Typkonzept in relationalen DBMS ist typischerweise sehr einfach, daher können Attribute mit der selben Domain verglichen werden, obwohl es möglicherweise semantisch keinen Sinn macht (z.B. MatrNr und VorlesungsNr)
  - …
- Von wem werden diese und andere Integritätsbedingungen überwacht…
  - … vom DBMS?
  - … vom Anwendungsprogramm?

Integritätsbedingungen

- Integritätsbedingungen sind Teil des Datenmodells
  - Wünschenswert ist eine zentrale Überwachung im DBMS innerhalb des Transaktionsmanagements
  - Einhaltung wäre 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 => ***deklarative Constraints***
- Für anwendungsspezifische Integritätsbedingungen ist häufig eine Definition und Realisierung im Anwendungsprogramm notwendig
  - Problem: Nur bei Verwendung des jeweiligen Anwendungsprogrammes ist die Einhaltung der Integritätsbedingungen garantiert sowie Korrektheit etc.
  - Meist: einfache Integritätsbedingungen direkt in DDL (deklarativ), Unterstützung für komplexere Integritätsbedingungen durch Trigger-Mechanismus => ***prozedurale Constraints***

Deklarative Constraints

- Deklarative Constraints sind Teil der Schemadefinition (`create table ...`)
- Arten:
  - Schlüsseleigenschaft: `primary key` (einmal), `unique` (beliebig)
    - `unique` kennzeichnet Schlüsselkandidaten
    - `primary key` kennzeichnet den Primärschlüssel
  - keine Nullwerte: `not null` (implizit bei `primary key`)
  - Typintegrität: Datentyp
  - Wertebedingungen: `check (<Bedingung>)`
  - referenzielle Integrität: `foreign key ... references ...` (nur Schlüssel)

Deklarative Constraints

- Constraints können
  - attributsbezogen (für jeweils ein Attribut)
  - tabellenbezogen (für mehrere Attribute)

  definiert werden.
- Beschreibung im Entwurf meist durch geschlossene logische Formeln der Prädikatenlogik 1.Stufe

  Beispiele
  - *Es darf keine zwei Räume mit gleicher R_ID geben.*

    $IB_1 : \forall r_1 \in Raum\ (\forall r_2 \in Raum\ (r_1[R\_ID] = r_2[R\_ID] \Rightarrow r_1 = r_2))$
  - *Für jede Belegung muss ein entsprechender Raum existieren.*

    $IB_2 : \forall b \in Belegung\ (\exists r \in Raum\ (b[R\_ID] = r[R\_ID]))$

Deklarative Constraints

- Umsetzung in SQL?
  - Bei IB_1 handelt es sich um eine Eindeutigkeitsanforderung an die Attributswerte von *R_ID* in der Relation *Raum* (Schlüsseleigenschaft).
  - IB_2 fordert die referenzielle Integrität der Attributswerte von *R_ID* in der Relation *Belegung* als Fremdschlüssel aus der Relation *Raum*.

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

CREATE TABLE belegung (
              b_id              number(10),
              r_id              varchar2(10)
              CONSTRAINT fk_belegung_raum         REFERENCES raum(r_id)
              (IB2)                               ...
);
```

Deklarative Constraints

- 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 $I \in IB$ auf dem aktuellen Datenbankzustand DB

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

Deklarative Constraints

- *Wann sollen Integritätsbedingungen geprüft werden?*
  - 1. Ansatz: Periodisches Prüfen der Datenbank *DB* gegen ***alle*** Integritätsbedingungen:

```
for each U <seit letztem Check> do
if (∀I ∈ IB: Check(I,U(DB))) 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

Deklarative Constraints

- 2. Ansatz: Inkrementelle Überprüfung bei jedem Update *U*
  - Voraussetzung: Update erfolgt auf einem konsistenten Datenbankzustand
  - dazu folgende Erweiterung:

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

dann:

```
<führe U durch>;
if (∀I ∈ IB: Check(I,U(DB))) then <ok>
else <rollback U>;
```

Deklarative Constraints

- Bei jedem Update U alle Integritätsbedingungen gegen die gesamte Datenbank zu testen ist immer noch zu teuer, daher Verbesserungen:
  1. Nur betroffene Integritätsbedingungen testen; z.B. kann die referenzielle Integritätsbedingung *Belegung* → *Raum*, nicht durch
     - Änderungen an der Relation *Dozent* verletzt werden
     - Einfügen in die Relation *Raum* verletzt werden
     - Löschen aus der Relation *Belegung* verletzt werden

     (siehe nächste Folien)

  2. Abhängig von U nur vereinfachte Form der betroffenen Integritätsbedingungen testen; z.B. muss bei Einfügen einer *Belegung* ***x*** nicht die gesamte Bedingung $\boldsymbol{IB_2}$ getestet werden, sondern es genügt der Test von:

     $$\exists r \in Raum\ (x[R\_ID] = r[R\_ID])$$

Deklarative Constraints

- *Was muss eigentlich geprüft werden?*

  Beispiel: Referentielle Integrität
  - Gegeben:
    - Relation R mit Primärschlüssel $\underline{\alpha}$ (potentiell zusammengesetzt)
    - Relation S mit Fremdschlüssel β (potentiell zusammengesetzt) aus Relation R
  - Referentielle Integrität ist erfüllt, wenn für alle Tupel $s \in S$ gilt

    1. $s.\beta$ enthält nur **null**-Werte oder nur Werte ungleich **null**

    und

    2. Enthält $s.\beta$ keine **null**-Werte, existiert ein Tupel $r \in R$ mit $s.\beta = r.\underline{\alpha}$
  - D.h.
    - Der Fremdschlüssel β in S enthält genauso viele Attribute wie der Primärschlüssel $\underline{\alpha}$ in R
    - Die Attribute haben dieselbe Bedeutung, auch wenn sie umbenannt wurden
    - ***Es gibt keine Verweise auf ein undefiniertes Objekt (dangling reference)***

      Das Tupel s in S wird hier auch ***abhängiger Datensatz*** (vom entsprechenden r in R) genannt

Deklarative Constraints

- Gewährleistung der Referentiellen Integrität
  - Es muss sichergestellt werden, dass keine dangling references eingebaut werden
  - D.h. für Relation R mit Primärschlüssel $\underline{\alpha}$ und Relation S mit Fremdschlüssel β aus R muss folgende Bedingung gelten:

    $$\pi_{\beta}(S) \subseteq \pi_{\underline{\alpha}}(R)$$

    (also alle gültigen Werte in β in S müssen auch in R vorkommen)
  - Erlaubte Änderungen sind also:
    1. Einfügen von Tupel s in S, wenn $s.\beta \in \pi_{\underline{\alpha}}(R)$
       (Fremdschlüssel β verweist auf ein existierendes Tupel in R)
    2. Verändern eines Wertes $w = s.\beta$ zu w', wenn $w' \in \pi_{\underline{\alpha}}(R)$
       (wie 1.)
    3. Verändern von $r.\underline{\alpha}$ in R nur, wenn $\sigma_{\beta=r.\underline{\alpha}}(S) = \varnothing$
       (es existieren keine Verweise in S auf Tupel r mit Schlüssel $\underline{\alpha}$ also keine abhängigen Tupel in S)
    4. Löschen von r in R nur, wenn $\sigma_{\beta=r.\underline{\alpha}}(S) = \varnothing$
       (wie 3.)

  Andernfalls: `ROLLBACK` der entspr. TA

Deklarative Constraints

– Beim Löschen in R weitere Optionen:

Option	Wirkung
`ON DELETE NO ACTION`	Änderungsoperation wird zurückgewiesen, falls abhängiger Datensatz in S vorhanden
`ON DELETE RESTRICT`	
`ON DELETE CASCADE`	Abhängige Datensätze in S werden automatisch gelöscht; kann sich über mehrstufige Abhängigkeiten fortsetzen
`ON DELETE SET NULL`	Wert des abhängigen Fremdschlüssels in S wird auf `null` gesetzt
`ON DELETE SET DEFAULT`	Wert des abhängigen Fremdschlüssels in S wird auf den Default-Wert der Spalte gesetzt

Deklarative Constraints

- Wertebedingungen (statische Constraints nach **`check`**-Klauseln)
  - Dienen meist zur Einschränkung des Wertebereichs
  - Ermöglichen die Spezifikation von Aufzählungstypen,

    z.B.
    ```
    create table Professoren (
                …
                Rang      character(2) check (Rang in (`W1´, `W2´, `W3´)),
                …
        )
    ```
  - Ermöglicht auch, die referentielle Integrität bei zusammen gesetzten Fremdschlüsseln zu spezifizieren (alle Teile entweder `null` oder alle Teile nicht `null`)
  - Achtung: **`check`**-Constraints gelten auch dann als erfüllt, wenn die Formel zu **`unknown`** ausgewertet wird (kann durch **`null`**-Wert passieren!!!)

    (Übrigens im Ggs. zu **`where`**-Bedingungen)

Deklarative Constraints

- Komplexere Integritätsbedingungen
  - In einer `check`-Bedingung können auch Unteranfragen stehen => IBs können sich auf mehrere Relationen beziehen (Verallgemeinerung der ref. Int.)
  - Beispiel:
    - Tabelle `pruefen` modelliert Relationship zwischen Student, Professor und Vorlesung
    - Das Constraint `VorherHoeren` garantiert, dass Studenten sich nur über Vorlesungen prüfen lassen können, die sie auch gehört haben

```
create table pruefen (
        MatrNr    integer references Studenten …
        VorlNr    integer references Vorlesungen …
        PersNr    integer references Professoren …
        Note      numeric(2,1) check (Note between 1.0 and 5.0),
        primary key (MatrNr, VorlNr)
        constraint VorherHoeren
                  check(    exists (  select * from hoeren h, pruefen p where
                                      h.VorlNr = p.VorlNr and h.MatrNr = p.MatrNr
                                    )
                  )
)
```

  - Diese IBs werden leider kaum unterstützt (Lösung: Trigger)

Prozedurale Constraints (Trigger)

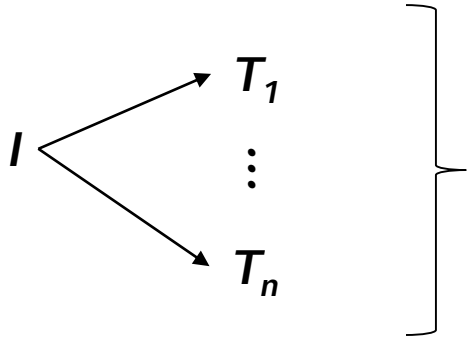
- Motivation: Komplexere Bedingungen als bei deklarativen Constraints und damit verbundene Aktionen wünschenswert.
- Trigger: Aktion (typischerweise PL/SQL-Programm), die einer Tabelle zugeordnet ist und durch ein bestimmtes Ereignis ausgelöst wird.
- Ein Trigger enthält Code, der die mögliche Verletzung 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/gelöschtem Datensatz ausgeführt.
- mögliche Zeitpunkte: vor (`BEFORE`) oder nach (`AFTER`) dem auslösenden Befehl

Prozedurale Constraints (Trigger)

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

  => Adressierung durch Präfix: `new.` bzw. `old.` (Syntax systemspezifisch)
- Befehlstrigger haben Zugriff auf die Änderungen durch die auslösenden Befehle (die typischerweise Tabellen verändern)

  => Adressierung durch `newtable` bzw. `oldtable` (Syntax systemspezifisch)
- Zu einer Integritätsbedingung *I* gehören in der Regel mehrere Trigger T_i



Je nach auslösendem Ereignis-Typ unterschiedliche Trigger für die Integritätsbedingung

Prozedurale Constraints (Trigger)

- Aufbau eines Trigger-Programms:

```
create or replace trigger <trig_name>
before/after/instead of  -- Trigger vor/nach/alternativ zu Auslöser ausführen
insert or update of <attrib1>, <attrib2>, ... or delete     -- Trigger-Ereignisse
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
...
begin
if inserting then <pl/sql Anweisungen>
end if;
if updating (<attrib1>) then <pl/sql Anweisungen>
end if;
if deleting then <pl/sql Anweisungen>
end if;
...                                         -- Code hier gilt für alle Ereignisse
end;
```

Prozedurale Constraints (Trigger)

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

$$\forall p \in Period\_Belegung\ (0 \leq p[Tag] \leq 6 \wedge p[Erster\_Termin] \leq p[Letzter\_Termin] \wedge Wochentag(p[Erster\_Termin]) = p[Tag] \wedge Wochentag(p[Letzter\_Termin]) = p[Tag])$$

Formulierung als deklaratives Constraint:

```
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)
);
```

Prozedurale Constraints (Trigger)

Formulierung als prozedurales Constraint (Trigger):

```
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
              (to_number(to_char(et, 'd')) != tag) or
              (to_number(to_char(lt, 'd')) != tag)
           then
              raise_application_error(-20089, 'Falsche Tagesangabe');
           end if;
        END;
```

Prozedurale Constraints (Trigger)

- Verwandtes Problem: Sequenzen für die Erstellung eindeutiger IDs

```
CREATE SEQUENCE <seq_name>
[INCREMENT BY n]                  -- Default: 1
[START WITH n]                    -- Default: 1
[{MAXVALUE n | NOMAXVALUE}]       -- Maximalwert (10^27 bzw. -1)
[{MINVALUE n | NOMINVALUE}]       -- Mindestwert (1 bzw. -10^26)
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];            -- Vorcachen, Default: 20
```

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

```
CREATE SEQUENCE seq_pers;
INSERT INTO Person (p_id, p_name, p_alter)
       VALUES (seq_pers.NEXTVAL, 'Ulf Mustermann', 28);
```

Prozedurale Constraints (Trigger)

- Beispiel mit Trigger:

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

INSERT INTO Person (p_name, p_alter)
        VALUES ('Ulf Mustermann', 28);
```

- Vorteil: Zuteilung der ID erfolgt transparent, d.h. kein expliziter Zugriff (über `.NEXTVAL`) in `INSERT`-Statement nötig!

Prozedurale Constraints (Trigger)

- Allgemeines Schema der Trigger-Abarbeitung

  Event *e* aktiviert während eines Statements *S* einer Transaktion eine Menge von Triggern $T = (T_1, \ldots, T_k)$

  1. Füge alle neu aktivierten Trigger $T_1, \ldots, T_k$ in die ***TriggerQueue*** *Q* ein
  2. Unterbreche die Bearbeitung von *S*
  3. Berechne `new` und `old` bzw. `newtable` und `oldtable`
  4. Führe alle `BEFORE`-Trigger in *T* aus, deren Vorbedingung erfüllt ist
  5. Führe die Updates aus, die in *S* spezifiziert sind
  6. Führe die `AFTER`-Trigger in *T* aus wenn die Vorbedingung erfüllt ist
  7. Wenn ein Trigger neue Trigger aktiviert, springe zu Schritt 1

Prozedurale Constraints (Trigger)

- Achtung:

  Eine (nicht-terminierende) Kettenreaktion von Triggern ist grundsätzlich möglich
- Eine Menge von Triggern heißt ***sicher*** (***safe***), wenn eine potentielle Kettenreaktion immer terminiert
  - Es gibt Bedingungen die hinreichend sind um Sicherheit zu garantieren (d.h. wenn sie erfüllt sind, ist die Trigger-Menge sicher, es gibt aber sichere Trigger-Mengen, die diese Bedingungen nicht erfüllen)
  - Typischerweise gibt es aber keine hinreichend und notwendigen Bedingungen, daher ist Sicherheit algorithmisch schwer zu testen.
- Eine Möglichkeit wäre wieder einen Abhängigkeits- (bzw. Aktivierungs-)graph
  - Knoten: Trigger
  - Kante von T_i nach T_j wenn die Ausführung von T_i T_j aktivieren kann
  - Keine Zyklen implizieren Sicherheit (Zyklen implizieren nicht notwendigerweise Unsicherheit)
  - ABER: ineffizient und nicht einfach zu realisieren (automatische Erkennung wann T_i T_j aktivieren kann?)

Prozedurale Constraints (Trigger)

- Trigger können auch noch für andere Aufgaben verwendet werden
  - Implementierung von Integritätsbedingungen und Erzeugung eindeutiger IDs (siehe dieses Kapitel)
  - Implementierung von Geschäftsprozessen (z.B. wenn eine Buchung ausgeführt wird, soll eine Bestätigungs-Email versandt werden)
  - Monitoring von Einfügungen/Updates (im Prinzip eine Kopplung der ersten beiden: wenn ein neuer Wert eingefügt wird, kann abhängig davon ein entsprechendes Ereignis ausgelöst werden)
  - Verwaltung temporär gespeicherter oder dauerhaft materialisierter Daten (z.B. materialisierte Views)