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

Skript zur Vorlesung:

Datenbanksysteme II

Sommersemester 2016

Kapitel 4
Datenintegrität und Datensicherheit

Vorlesung: Prof. Dr. Peer Kröger

http://www.dbs.ifi.lmu.de/cms/Datenbanksysteme_II

Dieses Skript basiert in Teilen auf den Skripten zur Vorlesung Datenbanksysteme II an der LMU München von

Prof. Dr. Christian Böhm (SoSe 2007),
PD Dr. Peer Kröger (SoSe 2008, 2014, 2015) und
PD Dr. Matthias Schubert (SoSe 2009)

4. Datenintegrität/-Sicherheit

4.1 Datenintegrität

4.2 Datensicherheit

4. Datenintegrität/-Sicherheit

4.1.1 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)

4.1.1 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?

4.1.1 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***

4. Datenintegrität/-Sicherheit

4.1.2 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)

4.1.2 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.*

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

4.1.2 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)                              ...
);
```

4.1.2 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*

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

4.1.2 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

4.1.2 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>;
```

4.1.2 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 IB_2 getestet werden, sondern es genügt der Test von:

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

4.1.2 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

4.1.2 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

4.1.2 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

4.1.2 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)

4.1.2 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, de 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)

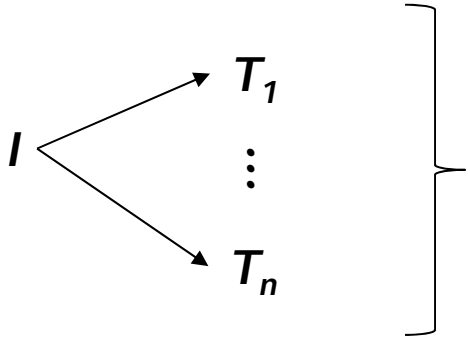
4. Datenintegrität/-Sicherheit

4.1.3 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

4.1.3 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

4.1.3 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;
```

4.1.3 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)
);
```

4.1.3 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;
```

4.1.3 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);
```

4.1.3 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!

4.1.3 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

4.1.3 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?)

4.1.3 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)

4. Datenintegrität/-Sicherheit

4.2.1 Einleitung

- Allgemeine Aspekte zum Datenschutz
  - Juristische und ethische Faktoren

    z.B. Bundes-Datenschutz-Gesetz, …
  - Organisations-(z.B. Firmen-)spezifische Regelungen

    z.B. Kreditkartenauskünfte, versch. Sicherheitsebenen für Abteilungen beim Geheimdienst, …
  - Technische Faktoren

    HW-Ebene, Betriebssystem-Ebene, DBMS-Ebene, …

4.2.1 Einleitung

- Datenschutzmechanismen
  - Identifikation und Authentisierung

    Benutzermanagement, …
  - Autorisierung und Zugriffskontrolle

    Regeln legen erlaubte Zugriffsarten von ***Sicherheitssubjekten*** auf ***Sicherheitsobjekten*** fest
    - Sicherheitssubjekt: aktive Entität, die Informationsfluss bewirkt, z.B. Benutzer(-gruppen), Anwendungsprogramme, Trigger, …
    - Sicherheitsobjekt: passive Entität mit Informationsinhalt(en), z.B. ein Tupel, ein Attribut, …
  - Auditing

    Buchführen über sicherheitsrelevante DB-Operationen

4.2.2 Einfache Zugriffskontrolle in SQL

- Discretionary Access Control (DAC)
  - Spezifiziert Regeln zum Zugriff auf Objekte
  - Eine Regel besteht aus
    - Einem Objekt (z.B. Relationen, Tupel, Attribute, …)
    - Einem Subjekt (z.B. Benutzer, Prozesse, …)
    - Einem Zugriffsrecht (z.B. „lesen“, „schreiben“, „löschen“, …)
    - Einem Prädikat, das eine Art Zugriffsfenster auf dem Objekt festlegt
    - Einem Booleschen Wert, der angibt, ob das Recht vom Subjekt an andere Subjekte weitergeben darf

4.2.2 Einfache Zugriffskontrolle in SQL

- Die Regeln werden typischerweise in einer eigenen Tabelle oder in einer Matrix (Spalten: Objekte, Zeilen: Subjekte) gespeichert
- Zugriff eines Subjekts auf ein Objekt nur, wenn entsprechender Eintrag in Tabelle/Matrix
- Umsetzung
  - Als View (mit den entsprechenden Update-Problematiken)
  - Abänderung der Anfrage entsprechend den Bedingungen
    - `select`-Klausel darf nur Attribute enthalten, auf die der entspr. Benutzer Zugriff hat
    - Zugriffsprädikat kann konjunktiv an die `where`-Bedingung angefügt werden
    - ...

4.2.2 Einfache Zugriffskontrolle in SQL

- Nachteile von DAC
  - Performanz: Abhängig von der Granularität der Autorisierung können diese Tabellen/Matrizen sehr groß werden
  - Beruht auf der Annahme, dass Erzeuger der Daten deren Eigner und damit für die Sicherheit verantwortlich ist
    - Erzeuger können Zugriffsrechte damit beliebig weitergeben
    - Beispiel Firma: Angestellte erzeugen Daten und sind dann in der Verantwortung für die Sicherheit dieser Daten
  - Weitergabe von Rechten kann zu Problemen führen
    - S1 gibt Recht an S2
    - S1 gibt Recht an S3
    - S2 gibt Recht an S3
    - S1 will Recht S3 wieder entziehen ???

4.2.2 Einfache Zugriffskontrolle in SQL

- Trotzdem:
  - DAC ist einfach umzusetzen und daher sehr gebräuchlich
  - Zugriffskontrolle im SQL-92 Standard basiert auf DAC-Modell
- Apropos: SQL Standard
  - Stellt keine Normen für Authentisierung oder Auditing auf
  - Einfache Zugriffskontrolle nach DAC-Modell
    - `grant` – vergibt Rechte
    - `revoke` – entzieht Rechte
    - Intial liegen alle Rechte beim Administrator (DBA)
- Manche DBMS stellen Zugriffskontroll-Mechanismen nach mächtigeren Modellen (z.B. dem MAC-Modell, siehe später) zur Verfügung

4.2.2 Einfache Zugriffskontrolle in SQL

- Autorisierung mit `grant`
  - Typische Form:

    ```
    grant <OPERATION> on <TABLE> to <USER>
    ```

  - Dabei ist `<OPERATION>`:
    - `select` Lesezugriff
    - `delete` Löschen
    - `insert (<Attribute>)` Einügen der spezifizierten Attribute
    - `update (<Attribute>)` Verändern der spezifizierten Attribute
    - `references (<Attribut>)` Fremdschlüssel auf das Attribut

      ACHTUNG: hier gilt es natürlich referentielle Integrität einzuhalten, daher könnte man dadurch die Schlüsselwerte der anderen Relation herausbekommen:
      - » Es gibt Relation `Agenten` mit Schlüssel = geheime Kennung
      - » Wir haben keine Zugriffsrechte auf diesen Schlüssel kennen aber das Schema von `Agenten`
      - » Mit `create table at(Kennung char(4) references Agenten);` können wir durch Einfügen einiger Zeilen prüfen ob entsprechende Werte in `Agenten` existieren

4.2.2 Einfache Zugriffskontrolle in SQL

- Recht zur Weitergabe von Rechten durch Anhängen von
  `with grant option`
  am Ende eines `grant`-Befehls
- Entziehen eines Rechts mit `revoke`
  - Bei Privileg mit Weitergaberecht:
    - `restrict` falls Weitergabe erfolgt, bricht DBMS mit Fehlermeldung ab
    - `cascade` löscht auch die Rechte, die durch Weitergabe entstanden sind
- Umsetzungen von bedingten Rechten in SQL durch Sichten
  Beispiel: Tutoren für EIP sollen nur die Daten der Studenten im ersten Semester lesen können

```
create view ErstSemester as
    select * from Studenten where Semester = 1;

grant select on ErstSemester to tutor
```

4. Datenintegrität/-Sicherheit

4.2.3 Verfeinerte Zugriffskontrolle

- Bisher nur
  - Explizite Autorisierung
    - Bei vielen Objekten viele Regeln => großer Aufwand
    - Schöner wäre, wenn wir uns durch ***implizite Autorisierung*** etwas sparen könnten
  - Positive Autorisierung
    - Darf ein Subjekt 4 der 5 möglichen Operationen auf einem Objekt, müssen alle 4 (explizit) erlaubt werden (analog: z.B. 1 aus einer Gruppe von 10 Subjekten hat als einziges Subjekt ein spez. Recht nicht, …)
    - Schöner wäre, z.B. *per default* alle zu erlauben und nur die eine Operation zu verbieten (***negative Autorisierung***)
    - Dazu nötig: Unterschied zwischen ***starker*** und ***schwacher Autorisierung:***

      Schwache Autorisierung wird als Standardeinstellung (z.B. für alle Subjekte) verwendet und gilt daher immer, falls keine andere (starke) Autorisierung erfolgt

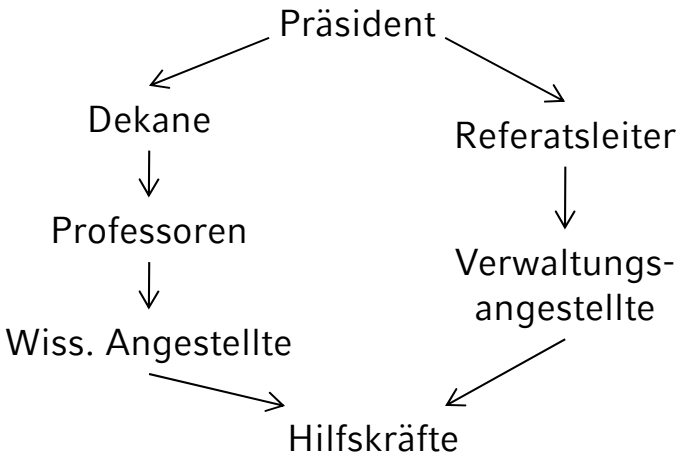
4.2.3 Verfeinerte Zugriffskontrolle

- Kernidee der Erweiterung
  - Subjekte, Objekte und Operationen werden hierarchisch angeordnet
  - Explizite Autorisierung auf einer bestimmten Stufe der Hierarchie bewirkt implizite Autorisierung auf anderen Stufen der Hierarchie
  - Unterscheidung in
    - Positive Autorisierung — schreibe (Objekt, Subjekt, Operation)
    - Negative Autorisierung — schreibe (Objekt, Subjekt, ¬Operation)
  - Unterscheidung zwischen
    - Starker Autorisierung — schreibe (…)
    - Schwacher Autorisierung — schreibe […]

4.2.3 Verfeinerte Zugriffskontrolle

- Implizite Autorisierung von Subjekten
  - Rolle
    - Funktion einer Menge von Benutzer im System
    - Beinhaltet die Rechte, die zur Umsetzung notwendig sind)
  - Rollenhierarchie enthält mind.
    - Eine ausgezeichnete Rolle mit der maximalen Menge an Rechten (z.B. DBA, Firmenleitung, …) als Wurzel der Hierarchie
    - Eine eindeutige grundlegende Rolle (z.B. alle Angestellten)

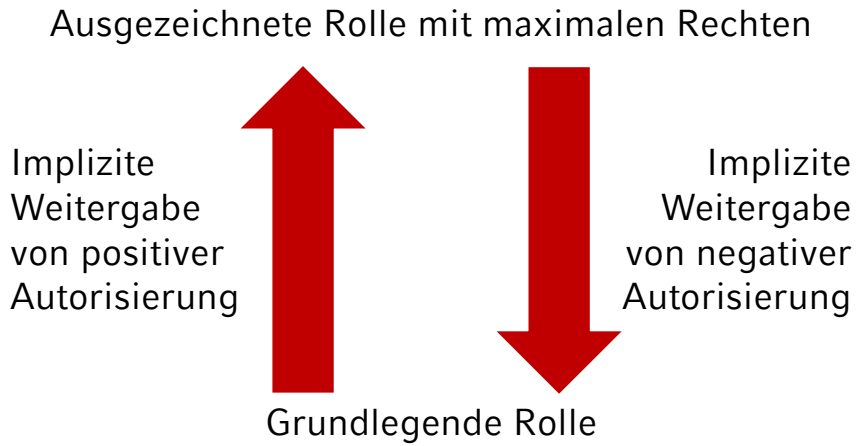
Beispiel:



4.2.3 Verfeinerte Zugriffskontrolle

- Regeln zur impliziten Autorisierung

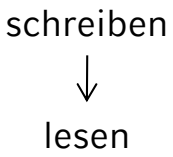
1. Eine explizite positive Autorisierung auf einer Stufe resultiert in einer impliziten positiven Autorisierung auf allen höheren Stufen (z.B. besitzen Dekane implizit alle Zugriffsrechte die explizit oder implizit für Professoren gelten)
2. Eine explizite negative Autorisierung auf einer Stufe resultiert in einer impliziten negativen Autorisierung auf allen niedrigeren Stufen (z.B. gilt der explizite Zugriffsverbot auf ein Objekt für den Referatsleiter implizit auch für den Verwaltungsangestellten)



4.2.3 Verfeinerte Zugriffskontrolle

- Implizite Autorisierung von Operationen
  - Analog: Operationshierarchien

    Beispiel

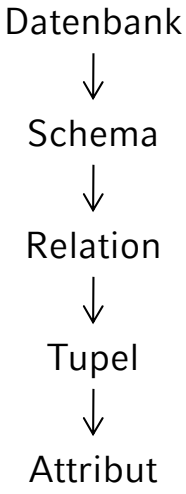


  - Weitergabe der Rechte nun umgekehrt:
    - Positive Autorisierung wird nach unten weitergegeben (Schreibberechtigung impliziert Leseberechtigung)
    - Negative Autorisierung wird nach oben weitergegeben (Leseverbot impliziert auch Schreibverbot)

4.2.3 Verfeinerte Zugriffskontrolle

- Implizite Autorisierung von Objekten
  - Granularitätshierarchien für Objekte

    Bsp.: Leserecht für eine Relation R sollte Leserecht für die einzelnen Tupel von R implizieren
  - Regeln hängen meist von der auszuführenden Operation ab, z.B.
    - Explizites Lese- und Schreibrecht auf einer Relation impliziert (nur) Leserecht auf deren Schema
    - Leserechte müssen immer auch nach unten implizit weiter geleitet werden
    - Definition einer neuen Relation hat keine Implikation auf andere Ebenen
    - ...



4.2.3 Verfeinerte Zugriffskontrolle

- Typhierarchien
  - Bieten eine weitere Dimension für implizite Autorisierung
  - Werden durch is-a-Beziehungen (Generalisierung/Spezialisierung) zwischen Entities definiert (vgl. oo Programmierung)
    - Zugriffsrecht auf einen Objekttypen O impliziert Zugriffsrecht auf von O vererbte Attribute im Untertypen
    - Attribut eines Untertypen ist nicht vom Obertypen erreichbar
    - Zugriff auf Objekttypen O impliziert Zugriff auf vom Obertypen ererbte Attribute in O
  - Problem:
    - Vererbung wird im relationalen Modell nicht unterstützt sondern nur simuliert
    - Daher wird eine implizite Autorisierung entlang einer Typhierarchie in relationalen DBMS meist nicht unterstützt

4. Datenintegrität/-Sicherheit

4.2.4 MAC und Multilevel DBs

- MAC = Mandatory Access Control
  - Einführung einer Sicherheitshierarchie

    z.B. „streng geheim“, „geheim“, „vertraulich“, „unklassifiziert“
  - Sicherheitseinstufung für
    - Subjekte s: clear(s) spezifiziert die Vertrauenswürdigkeit von s
    - Objekte o: class(o) spezifiziert die Sensitivität von o
  - Typische Zugriffsregeln:
    1. Subjekt s darf Objekt o nur lesen, wenn class(o) ≤ clear(s)
    2. Objekt o wird mit mindestens der Einstufen des schreibenden Subjekts s versehen, d.h. clear(s) ≤ class(o)
    - Bemerkungen
      - Die zweite Regel stellt sicher, dass ein Benutzer der Klasse „streng geheim“ auch nur „streng geheime“ Objekte schreibt, insbesondere v.a. keine „unklassifizierten“ Objekte (Write Down , Leak)
      - Niedrig eingestufte Objekte können „hochklassifiziert“ werden (und sehen dadurch möglicherweise ungewollt vertrauenswürdiger /fundierter aus)

4.2.4 MAC und Multilevel DBs

- Bewertung von MAC
  - Potentiell größere Sicherheit durch mächtigeren (ausdrucksstärkeren) Kontrollmechanismus
  - Typischerweise Organisationsproblem
    - Benutzer unterschiedlicher Klassifikationsstufen können nicht zusammen arbeiten
    - Alle Objekte der Datenbank müssen eingestuft sein
  - Problem des Abgriffs nicht freigegebener Daten besteht immer noch:

    Relation Agenten

Class (Tupel)	Kennung	class	Name	class	Drink	class
sg sg	007 008	g sg	James Bond Harry Potter	g sg	Wodka Limo	sg sg

    Benutzer mit clear = g sieht

Kennung	Name	Drink
007	James Bond	---

    und möchte Tupel mit Kennung 008 eingeben, was verweigert wird (womit klar ist, dass diese Kennung schon existiert, das entspr. Tupel aber höher klassifiziert ist)

4.2.4 MAC und Multilevel DBs

- Multilevel DBs
  - Lösung des Problems durch ***Polyinstanziierung***
    - Ein Tupel darf mehrfach mit unterschiedlichen Sicherheitseinstufungen vorkommen (alle Einstufungen müssen aber tatsächlich paarweise verschieden sein)
    - Die DB stellt sich damit Nutzern unterschiedlicher Einstufungen unterschiedlich dar

      (Im Beispiel von vorher gäbe es nun zwei Einträge mit Schlüssel 008 mit unterschiedlicher Klassifizierung)
  - Damit können nun auch Benutzer unterschiedlicher Klassifikationen auf „den gleichen“ Daten arbeiten, da eine Bearbeitung nicht sofort zu einer Höherklassifizierung (nach Regel 2 des MAC-Modells) führt

4.2.4 MAC und Multilevel DBs

- Umsetzung
  - Schema einer Multilevel Relation *R* besteht aus
    - *n* Attributen A_i mit ihren Domänen D_i (wie gehabt)
    - Für jedes Attribut A_i eine Klassifizierung C_i
    - Eine Klassifizierung TC des gesamten Tupels
  - Für jede Zugriffsklasse c gibt es dann eine ***Relationeninstanz*** R_c
  - In R_c sind alle Tupel $(a_1,c_1, a_2,c_2, \ldots a_n,c_n, tc)$ mit $c \geq c_i$
  - Der Wert a_1 eines Attribut A_1 ist sichtbar (d.h. $a_1 \in D_i$) falls $c \geq c_i$ ansonsten `null`
- Integritätsbedingungen
  - Fundamental im normalen relationalen Modell:
    - Eindeutigkeit des Schlüssels
    - Referentielle Integrität
  - In Multilevel DBs Erweiterung nötig

4.2.4 MAC und Multilevel DBs

- Schlüssel:

  in Multilevel Relationen heißt der benutzerdefinierte Schlüssel ***sichtbarer Schlüssel***

  Sei im folgenden K der sichtbare Schlüssel von Relation *R*

- Entity Integrität

  für alle Instanzen R_c von *R* und alle Tupel $r \in R_c$ gilt

  1. $A_i \in K \Rightarrow r.A_i \neq \texttt{null}$

     d.h. kein Schlüsselattribute besitzt `null`-Werte

  2. $A_i, A_j \in K \Rightarrow r.C_i = r.C_j$

     d.h. alle Schlüssel haben die gleiche Klassifizierung (sonst kann Möglichkeit des Zugriffs auf Tupel nicht eindeutig geklärt werden)

  3. $A_i, \notin K \Rightarrow r.C_i \geq r.C_K$ (wobei C_K Zugriffsklasse des Schlüssels)

     d.h. Nicht-Schlüsselatribute haben mindestens die Zugriffsklasse des Schlüssels

4.2.4 MAC und Multilevel DBs

- Null-Integrität

  für alle Instanzen R_c von *R* gilt

  1. Für alle Tupel $r \in R_c$ gilt: $r.A_i = \texttt{null} \Rightarrow r.C_i = r.C_K$

     d.h. `null`-Werte erhalten die Klassifizierung des Schlüssels

  2. R_c ist subsumierungsfrei, d.h. es existieren keine zwei Tupel r und s in R_c, bei denen für alle Attribute A_i entweder

     - $r.A_i = s.A_i$ und $r.C_i = s.C_i$

     oder

     - $r.A_i \neq \texttt{null}$ und $s.A_i = \texttt{null}$

     gilt

     d.h. Tupel, über die schon mehr bekannt ist, werden „verschluckt“

4.2.4 MAC und Multilevel DBs

Beispiel für Subsumtionsfreiheit:

Class (Tupel)	Kennung	class	Name	class	Drink	class
g sg	007 008	g sg	James Bond Harry Potter	g sg	--- Limo	g sg

Ein streng geheimer Benutzer fügt Drink von 007 ein

⇒ Er erwartet dann

Class (Tupel)	Kennung	class	Name	class	Drink	class
sg sg	007 008	g sg	James Bond Harry Potter	g sg	Wodka Limo	sg sg

⇒ Ohne Subsumtionsfreiheit

Class (Tupel)	Kennung	class	Name	class	Drink	class
g sg sg	007 007 008	g g sg	James Bond James Bond Harry Potter	g g sg	--- Wodka Limo	g sg sg

4.2.4 MAC und Multilevel DBs

– Interinstanz-Integrität

Die Konsistenz ziwschen den einzelnen Instanzen der Multilevel-Relation muss gewährleistet sein.

Daher: für alle Instanzen R_c und $R_{c'}$ von R mit $c < c'$ gilt

$$R_{c'} = f(R_c, c')$$

wobei die Filterfunktion f wie folgt arbeitet:

1. Für jedes $r \in R_c$ mit $r.C_K \leq c'$ muss ein Tupel $s \in R_{c'}$ existieren, mit

$$s.A_i = \begin{cases} r.A_i & \text{wenn } r.C_i \leq c' \\ \text{null} & \text{sonst} \end{cases} \quad \text{und} \quad s.C_i = \begin{cases} r.C_i & \text{wenn } r.C_i \leq c' \\ r.C_K & \text{sonst} \end{cases}$$

2. $R_{c'}$ enthält außer diesen keine weiteren Tupel
3. Subsumierte Tupel werden eliminiert

4.2.4 MAC und Multilevel DBs

- Polyinstanziierungs-Integrität

  für alle Instanzen R_c von *R* und alle A_i gilt die funktionale Abhängigkeit:

  $$\{K, C_K, C_i\} \rightarrow A_i$$

  d.h. ein Tupel ist eindeutig bestimmt durch den Schlüssel und die Klassifizierung aller Attribute

  (diese Bedingung entspricht der „normalen“ Schlüsselintegrität)

- Umsetzung
  - Zerlegung einer Multilevel-Relation in mehrere normale Relationen (für jede Klassifizierungsebene), die bei Benutzeranfragen entsprechend zusammengesetzt werden können
  - Sicherstellung der hier genannten Integritätsbedingungen meist durch Trigger