Kapitel 11

Dr. Jérôme Kunegis

Datenintegrität, Views und Zugriffsrechte

WeST –
Web Science & Technologien

Lernziele

Verankerung von Integritätsregeln in DB

$\rightarrow$ effektivere Integritätssicherung

$\rightarrow$ einfachere Anwendungsprogrammierung

Durch:

- Integritätsbedingungen
  - Constraints und Assertions
  - Triggers
- Views
- Zugriffskontrolle

Typen der Integritätsbedingungen

Statische Bedingungen

- Invarianten des Datenmodells (Primärschlüssel, Fremdschlüssel)
- Anwendungsspezifische Anforderungen
  - bzgl. eines Attributes im Tupel
  - bzgl. mehrerer Attribute im Tupel
  - bzgl. mehrerer Tupel der Relation
  - bzgl. mehrerer Relationen

Dynamische Bedingungen

Zeitpunkt der Überprüfung:

- Ende des SQL-Befehls
- Ende der Transaktion

Reaktion auf Verletzung:

- Nichtausführung bzw. Rückgängigmachen der Anweisung
- Abbruch der Transaktion
- Ausführung von Folgeänderungen

Integritätsbedingungen: Beispiele

Statische Bedingungen:

- Jeder Student hat eine Matrikelnummer
- Alle Noten liegen zwischen 1 und 5
- Ein Seminar kann von maximal 20 Teilnehmern belegt werden
- Zwei Veranstaltungen können nicht zeitlich überlappend im selben Raum stattfinden

Dynamische Bedingungen:

- Die Prüfung in einem bestimmten Fach kann von dem Teilnehmer maximal dreimal wiederholt werden
- Status "teilgenommen" wechselt nur nach "bestanden" oder "nicht bestanden"
- Status "bestanden" kann nicht mehr geändert werden

Ausdrucksmittel zur Spezifikation von Integritätsbedingungen

- Constraints
- Assertions

} nur statische Integritätsbedingungen

- Triggers

Syntax von Constraints und Assertions

CREATE TABLE tablename
({ colname datatype [**DEFAULT** …] [{ colconstraint …}] …}
[{tabconstraint …}])

colconstraint ::= **NOT NULL** |
[**CONSTRAINT** constrname]
{ **UNIQUE** | **PRIMARY KEY** | **CHECK** (searchcond) |
REFERENCES tablename [(colname)] […] }

tabconstraint ::=
[**CONSTRAINT** constrname]
{ **UNIQUE** colname {, colname …} |
PRIMARY KEY colname {, colname …} |
CHECK (searchcond) |
FOREIGN KEY colname {, colname …}
REFERENCES tablename [(colname)] […] }

CREATE ASSERTION assertname **CHECK** (searchcond)
[**INITIALLY** {**DEFERRED** | **IMMEDIATE**}]

Semantik von Constraints und Assertions

Semantik von **CHECK** (searchcond)
mit sql2trc[searchcond] = $F(t_1, ..., t_n)$ mit freien Variablen $t_1, ..., t_n$:

$$\forall t_1 \ldots \forall t_n \, (\, (\, t_1 \in R_1 \wedge \ldots \wedge t_n \in R_n \,) \Rightarrow F(t_1, ..., t_n) \,) \quad \Big\} \; I_j$$

Semantik der Datenbank aus logischer Sicht: $H \wedge I_1 \wedge \ldots \wedge I_m$
mit elementarer Formel (Fakten) H und Integritätsbed. $I_1, ..., I_m$

Mögliche Implementierung:
bei potentieller Verletzung von Ij Ausführen von

SELECT t1, ..., tn **FROM** R1, ..., Rn **WHERE NOT** searchcond;

und Test auf Leerheit des Resultats

Zeitpunkt der Integritätsprüfung:
nach der SQL-Anweisung (... **IMMEDIATE**) oder
am Ende der Transaktion (... **DEFERRED**)

Reaktion bei Integritätsverletzung: Rollback

Statische Integritätsbedingungen: Beispiele

Kandidatenschlüssel: **UNIQUE**
Primärschlüssel: **PRIMARY KEY**
Fremdschlüssel: **FOREIGN KEY**

Wertebereichseinschränkungen
... **CHECK** (Semester **BETWEEN** 1 **AND** 13)
Aufzählungstypen
... **CHECK** (Rang **IN** ('C2', 'C3', 'C4'))

Constraints: Beispiel

Constraints der Relation 'Studenten':

- Matrikelnummer ist für jede Person eindeutig identifizierend und soll in der Relation als Primärschlüssel verwendet werden.
- Versicherungsnummer ist ebenfalls eindeutig identifizierend; jede Versicherungsnummer darf in der Relation nur einmal vorkommen.
- Angaben über den Namen dürfen nicht fehlen (d.h. Nullwerte sind in diesem Attribut nicht zulässig).
- Angaben über Semester dürfen nicht fehlen (d.h. Nullwerte sind in diesem Attribut nicht zulässig).
- Es gibt nur Semester zwischen 1…13.
- Neue Studenten sind immer dem 1. Semester zuzuordnen, sofern nicht ein anderes Semester explizit angegeben wurde

Constraints: Beispiel (2)

Die Constraints der Relation 'Studenten'
in der entsprechenden SQL-Anweisung:

```
CREATE TABLE Studenten
( MatrNr    integer PRIMARY KEY,
  VersNr    integer UNIQUE,
  Name      varchar(30) NOT NULL,
  Semester integer DEFAULT 1
                    NOT NULL
                    CHECK (Semester BETWEEN 1 AND 13) )
```

Constraints: Beispiel (2)

Hinweis:
"attributspezifische" Constraints können sich nur auf das einzelne Attribut beziehen, bei deren Definition sie angegeben sind

Die folgende Form ist z.B. in Oracle nicht zulässig:

```
CREATE TABLE Studenten
( MatrNr      integer PRIMARY KEY,
  VersNr      integer UNIQUE CHECK (Semester BETWEEN 1 AND 13) ),
  Name        varchar(30) NOT NULL,
  Semester integer DEFAULT 1 NOT NULL   );
```

Constraints: Beispiel (3)

```
CREATE TABLE Studenten
        ( MatrNr        integer PRIMARY KEY,
     VersNr      integer UNIQUE,
        Name            varchar(30) NOT NULL,
        Semester integer NOT NULL
    CONSTRAINT  SemCheck CHECK (Semester BETWEEN 1 AND 13)    );

ALTER TABLE Studenten DROP CONSTRAINT SemCheck;
```

Constraints: Beispiel (4)

```
CREATE TABLE Studenten
        ( MatrNr integer,  VersNr integer, Name varchar(30),  Semester integer,
    PRIMARY KEY (MatrNr),
        CONSTRAINT s1 CHECK (Name IS NOT NULL),
    CONSTRAINT      s2 UNIQUE (VersNr),
        CONSTRAINT s3 CHECK (Semester IS NOT NULL),
        CONSTRAINT s4 CHECK (Semester BETWEEN 1 AND 13)    );

  ALTER TABLE Studenten DROP CONSTRAINT s4;
  ALTER TABLE Studenten ADD CONSTRAINT s4
      CHECK (Semester BETWEEN 1 AND 11);
```

Constraints: Beispiel (5)

```
CREATE TABLE Studenten
        ( MatrNr        integer PRIMARY KEY,
          Name          varchar(30) NOT NULL,
          Semester      integer NOT NULL CHECK (Semester BETWEEN 1 AND
13));

CREATE TABLE Professoren
        ( PersNr        integer PRIMARY KEY,
          Name          varchar(30) NOT NULL,
      Rang              character(2) CHECK (Rang IN ('C2', 'C3', 'C4')),
          Raum          integer UNIQUE );
```

Assertions: Beispiel

CREATE TABLE Professoren (…)

CREATE TABLE Studenten (…)

"Professoren sind keine Studenten":

CREATE ASSERTION ProfNotStudent

CHECK (**NOT EXISTS**

(**SELECT** AusweisNr **FROM** Professoren

  INTERSECT

  SELECT AusweisNr **FROM** Studenten));

Referentielle Integrität: Motivation

Fremdschlüssel

verweisen auf Tupel einer Relation

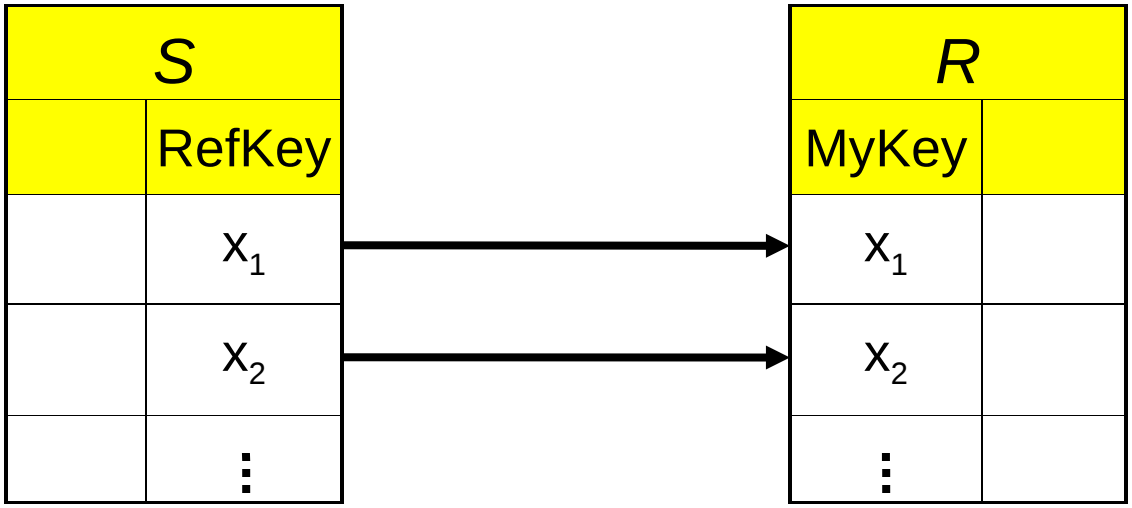
z.B. *Vorlesungen.gelesenVon* verweist auf Tupel in Professoren

Referentielle Integrität

Fremdschlüssel müssen auf existierende Tupel verweisen oder NULL enthalten

Einhaltung referentieller Integrität

Originalzustand



Mögliche Änderungsoperationen:

UPDATE *R*

SET MyKey = x_5

WHERE MyKey = x_1

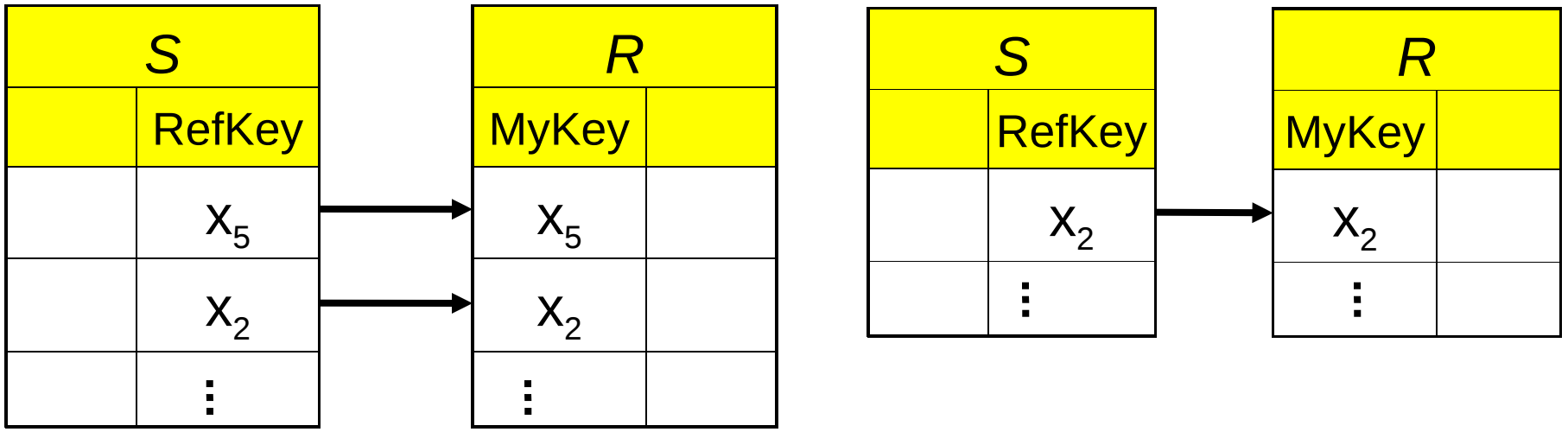
;

DELETE FROM *R*

WHERE MyKey = x_1

;

Kaskadieren



CREATE TABLE *S*

(...,

RefKey **integer REFERENCES** *R*

ON UPDATE CASCADE);

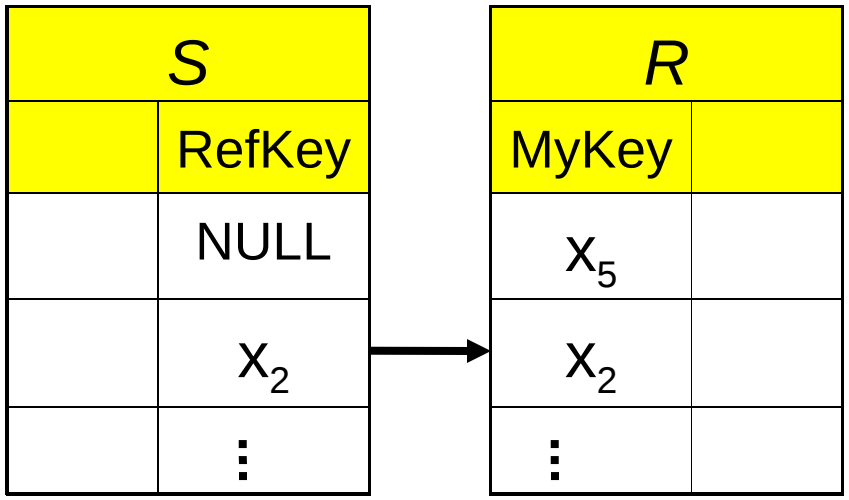
CREATE TABLE *S*

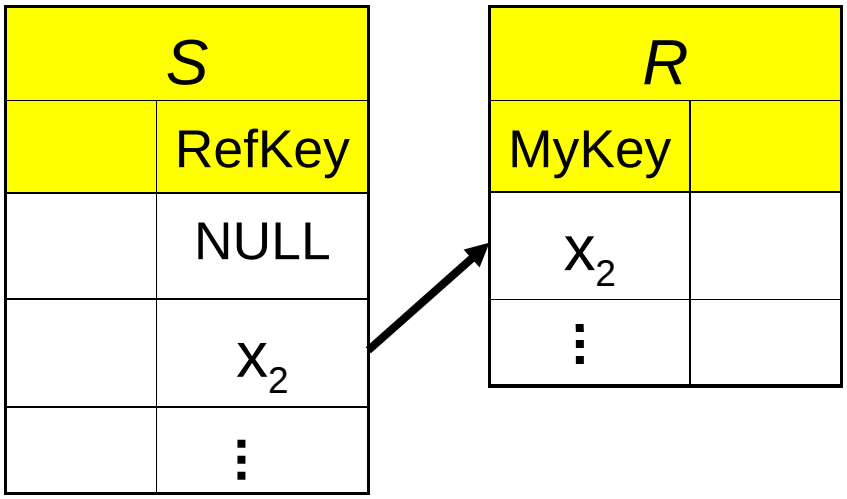
(...,

RefKey **integer REFERENCES** *R*

ON DELETE CASCADE);

Auf NULL bzw. DEFAULT-Wert setzen





CREATE TABLE *S*

(...,

RefKey **integer REFERENCES** *R*

ON UPDATE SET NULL);

CREATE TABLE *S*

(...,

RefKey **integer REFERENCES** *R*

ON DELETE SET NULL);

Referentielle Integrität in SQL: Motivation

CREATE TABLE *R*
(MyKey **integer PRIMARY KEY**, …);

CREATE TABLE *S*
(...,
RefKey **integer REFERENCES** *R (MyKey)*);

Optionen für Fremdschlüsselbedingungen

Grobsyntax:

... **FOREIGN KEY** (column {, column …})
REFERENCES [user .] table [(column {, column …})]
[**ON DELETE** {**NO ACTION** | **CASCADE** | **SET NULL** | **SET DEFAULT**}]
[**ON UPDATE** {**NO ACTION** | **CASCADE** | **SET NULL** | **SET DEFAULT**}]
[**INITIALLY** {**IMMEDIATE** | **DEFERRED**}] …

Semantik von
CREATE TABLE R ...
FOREIGN KEY A1, …, Am **REFERENCES** R1 (B1), …, Rm (Bm):

$$\forall t\,(\, t \in R \Rightarrow (\,(t.A1=NULL \wedge \ldots \wedge t.Am=NULL) \vee (\exists t_1 \ldots \exists t_m\,(t_1 \in R1 \wedge \ldots \wedge t_m \in Rm \wedge t_1.B1=t.A1 \wedge \ldots \wedge t_m.Bm=t.Am))\,)\,)$$

Reaktion bei Integritätsverletzung:

- Zurückweisen der Löschung/Änderung (bei **NO ACTION**)
- Löschen/Ändern aller "abhängigen" Tupel (bei **CASCADE**)
- Fremdschlüssel in "abhängigen" Tupeln auf Default-/Nullwert setzen

Referentielle Integrität: Beispiel (1)

CREATE TABLE hören

(MatrNr **integer REFERENCES** Studenten(MatrNr) **ON DELETE CASCADE**,

VorlNr **integer REFERENCES** Vorlesungen(VorlNr) **ON DELETE CASCADE**,

PRIMARY KEY (MatrNr, VorlNr));

"Anmeldungen der Gasthörer zu Vorlesungen":

CREATE TABLE gasthören

(MatrNr **integer**, VorlNr **integer**, …

PRIMARY KEY (MatrNr, VorlNr),

FOREIGN KEY (MatrNr, VorlNr) **REFERENCES** hören (MatrNr,VorlNr)

ON DELETE CASCADE);

Referentielle Integrität: Beispiel (2)

```
CREATE TABLE Assistenten
        ( PersNr            integer PRIMARY KEY,
          Name              varchar(30) NOT NULL,
          Fachgebiet        varchar(30),
          Boss              integer,
          FOREIGN KEY       (Boss) REFERENCES Professoren (PersNr)
          ON UPDATE CASCADE
          ON DELETE SET NULL);
```

Hinweis: "ON UPDATE CASCADE" wird nicht von allen Datenbank-Implementierungen unterstützt. (Bei Oracle z.B. nur manche Tabellen-Typen!)

Syntax und Semantik von CREATE TRIGGER

Grobsyntax:

CREATE TRIGGER trigger-name {**BEFORE** | **AFTER** | **INSTEAD OF**}
{ **DELETE** | **INSERT** | **UPDATE** [**OF** column {, column …}] }
ON table [**REFERENCING OLD AS** corrvar **NEW AS** corrvar]
[**FOR EACH ROW** | **FOR EACH STATEMENT**]
[**WHEN** (condition)] (statement-sequence)

Semantik mit sql2trc[condition] = F:

Werte F bei spez. Ereignis aus und starte Aktion, falls F wahr

- Fall 1 (**FOR EACH STATEMENT**):
  F hat keine freien Variablen
- Fall 2 (**FOR EACH ROW**):
  F hat eine oder zwei freie Variablen – told und tnew –, die an den alten bzw. neuen Wert des jeweiligen Tupels gebunden werden

Datenbank-Trigger: Beispiel (1)

"keine Verringerung von Semestern":

```
CREATE TRIGGER SemesterKontrolle
BEFORE UPDATE OF Semester ON Studenten
FOR EACH ROW
REFERENCING OLD AS old, NEW AS new
WHEN (old.Semester > new.Semester)
(ROLLBACK WORK);
```

Datenbank-Trigger: Beispiel (2)

"keine Verringerung von Semestern" in Oracle Syntax

```
CREATE TRIGGER SemesterKontrolle
BEFORE UPDATE OF Semester ON Studenten
FOR EACH ROW
WHEN (:old.Semester > :new.Semester)
BEGIN
  ROLLBACK;
END;
```

Datenbank-Trigger: Beispiel (3)

"keine Degradierung von Professoren" in Oracle Syntax

```
CREATE TRIGGER ProfKontrolle
BEFORE UPDATE OF Rang ON Professoren
FOR EACH ROW
WHEN (old.Rang IS NOT NULL)
BEGIN
        if :old.Rang = 'C3' and :new.Rang = 'C2' then
                :new.Rang := 'C3';
        end if;
        if :old.Rang = 'C4' then
                :new.Rang := 'C4'
    end if;
    if :new.Rang is NULL then
                :new.Rang := :old.Rang;
    end if;
END;
```

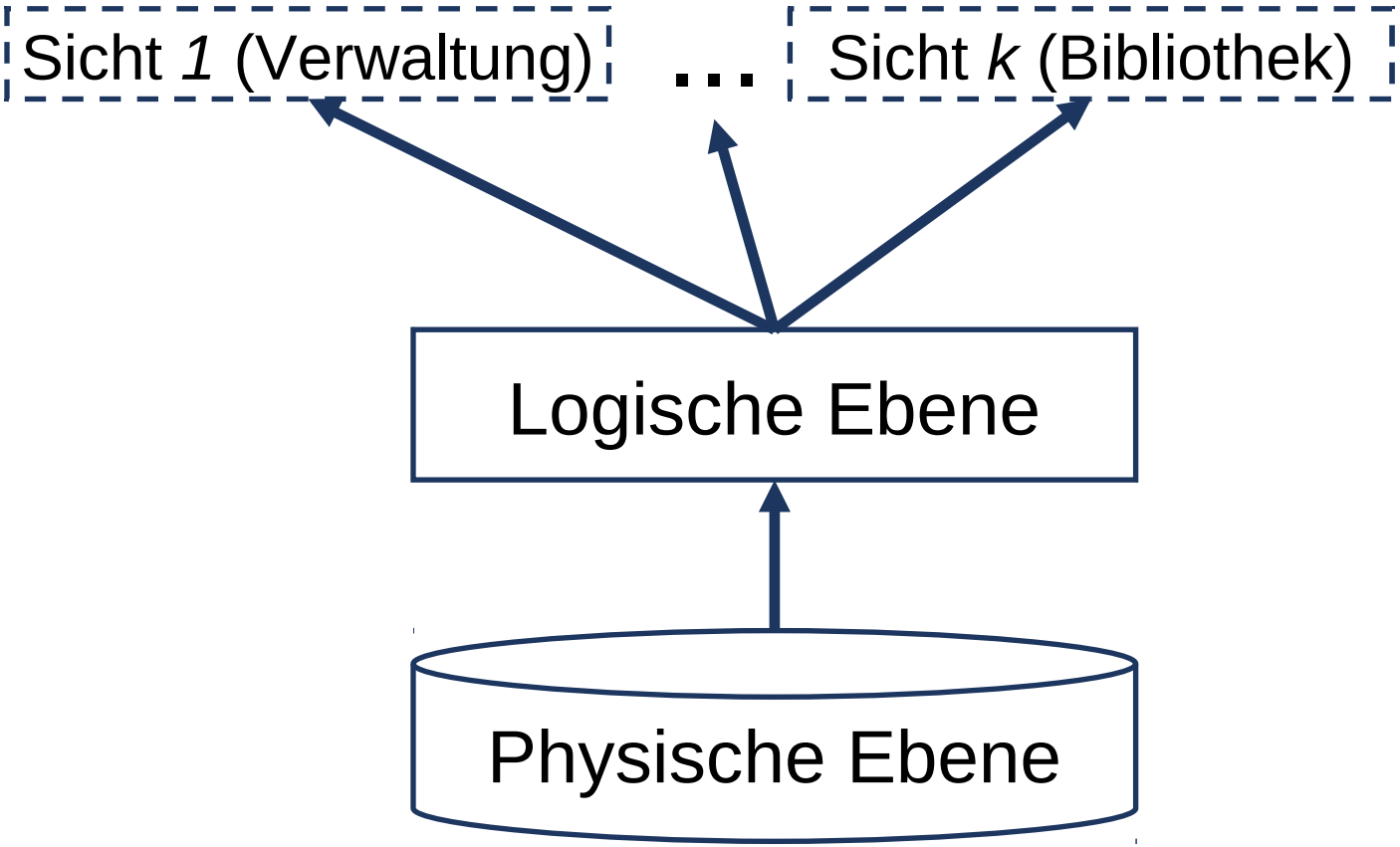
Trigger: eine Problemquelle?

Effekte von Triggern (z.B. Updates) können in Folge weitere Trigger auslösen!

Nebeneffekte der verketteten Trigger-Ausführung z.T. abhängig von der Reihenfolge

Ergebnis der komplexen Verkettungen oft schwer vorhersehbar

Views (Sichten, Virtuelle Relationen)



Ziel: simplifizieren Integritätsssicherung, Anfragen und Schema-Änderungen

Views (Sichten, Virtuelle Relationen)

Grobsyntax:
CREATE VIEW view-name [(column {, column …})]
AS select-block [**WITH CHECK OPTION**]

Für **CREATE VIEW** MyView **AS** viewquery ist
sql2ra [**SELECT** A1, …, Am
FROM TAB1, …, TABk, MyView
WHERE F]
$= \pi[A1, ..., Am] (sql2ra [F] (TAB1 \times \ldots \times TABk \times sql2ra[viewquery]))$

Sichten: Verwendung

… für den Datenschutz (ausblenden gewisser Attribute)

```
CREATE VIEW prüfenSicht as
        SELECT MatrNr, VorlNr, PersNr
        FROM prüfen;
```

… statistische Sicht

```
CREATE VIEW PrüfGüte(Name, GüteGrad) AS
      (SELECT p.Name, AVG(p.Note)
       FROM Professoren p, prüfen e
                  WHERE p.PersNr = e.PersNr
       GROUP BY p.Name, p.PersNr
       HAVING COUNT(*) > 50);
```

Sichten: Verwendung

… für die Vereinfachung von Anfagen:

```
CREATE VIEW StudProf (Sname, Semester, Titel, Pname) as
        SELECT s.Name, s.Semester, v.Titel, p.Name
        FROM Studenten s, hören h, Vorlesungen v, Professoren
p
        WHERE s.Matr.Nr=h.MatrNr and h.VorlNr=v.VorlNr and
            v.gelesenVon = p.PersNr ;

SELECT DISTINCT Semester
FROM StudProf
WHERE PName='Sokrates' ;
```

Sichten zur Modellierung von Generalisierung (Vertikale Partitionierung)

```
CREATE TABLE Angestellte
        (PersNr integer NOT NULL,
         Name   varchar (30) not null);

CREATE TABLE ProfDaten
        (PersNr integer NOT NULL,
         Rang   character(2),
         Raum   integer);

CREATE TABLE AssiDaten
        (PersNr  integer NOT NULL,
         Fachgebiet      varchar(30),
         Boss            integer);
```

Sichten zur Modellierung von Generalisierung (Vertikale Partitionierung) (2)

```
CREATE VIEW Professoren AS
        SELECT *
        FROM Angestellte a, ProfDaten d
        WHERE a.PersNr = d.PersNr;

CREATE VIEW Assistenten AS
        SELECT *
        FROM Angestellte a, AssiDaten d
        WHERE a.PersNr=d.PersNr;
```

➔ Untertypen als Views

Sichten zur Modellierung von Generalisierung (Horizontale Partitionierung)

```
CREATE TABLE Professoren
        (PersNr         integer not null,
        Name            varchar (30) not null,
        Rang            character (2),
        Raum            integer);

CREATE TABLE Assistenten
        (PersNr         integer not null,
        Name            varchar (30) not null,
        Fachgebiet      varchar (30),
        Boss            integer);

CREATE TABLE AndereAngestellte
        (PersNr         integer not null,
        Name            varchar (30) not null);
```

Sichten zur Modellierung von Generalisierung (Horizontale Partitionierung) (2)

```
CREATE VIEW Angestellte AS
        (SELECT PersNr, Name
        FROM Professoren)
          UNION
        (SELECT PersNr, Name
        FROM Assistenten)
         UNION
        (SELECT PersNr, Name
        FROM AndereAngestellte);
```

➔ Obertyp als View

Views zur Maskierung von Schema-Änderungen

1) bisherige Anfrage:
   SELECT * **FROM** Professoren **WHERE** PersNr = 1234

2) Schema-Änderung (plus Datenbank aktualisieren oder neu laden):

   a) **ALTER TABLE** Professoren
   ADD Vorname VARCHAR(20), Nachname VARCHAR(20)

   b) **UPDATE** Professoren SET Vorname = SUBSTR (Name, ...),
   Nachname = SUBSTR (Name, ...)

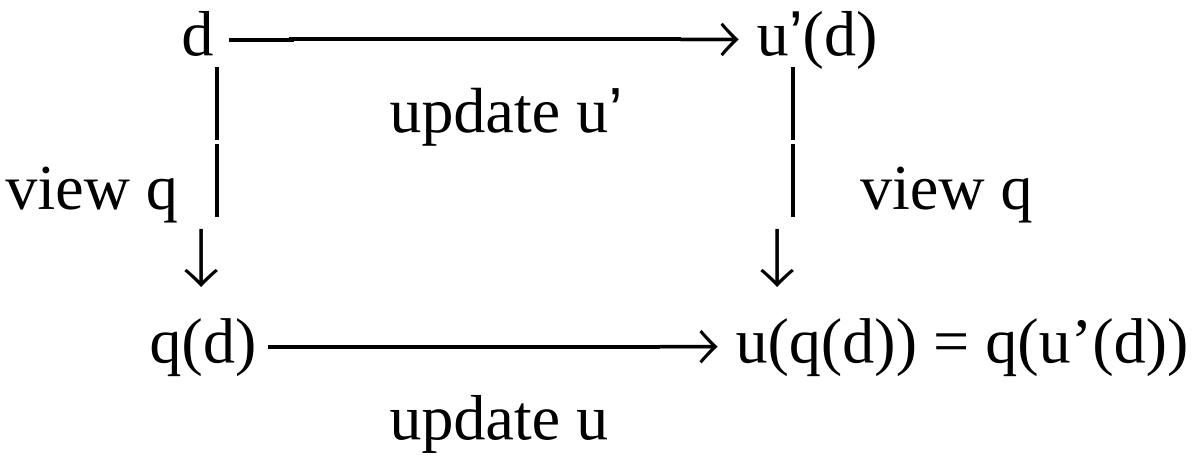
   c) **ALTER TABLE** Professoren **DROP** Name

3) Vorgehen zur "Bewahrung" der bisherigen Anfrage:
   a) Professoren in ProfessorenDaten umbenennen
   b) **CREATE VIEW** Professoren (PersNr, Name, Rang, Raum) **AS**
   SELECT PersNr, **CONCAT**(Vorname , ' ', Nachname), Rang, Raum
   FROM ProfessorenDaten

Updates auf Views

Sei D die Menge aller Datenbanken.
Views und Updates sind partielle Funktionen q: D $\rightarrow$ D, u: D $\rightarrow$ D.

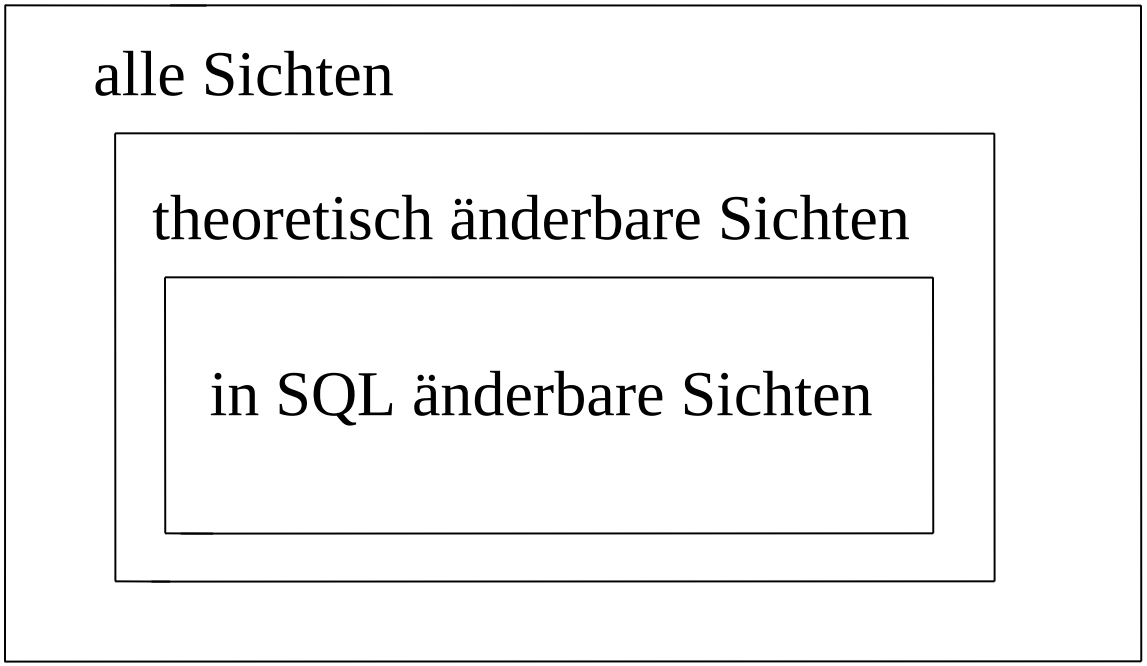
Eine View q heißt *änderbar* genau dann,
wenn es für jede Datenbank d, auf der q definiert ist,
und jeden auf q(d) definierten Update u
einen eindeutigen Update u' gibt,
der auf d definiert ist und für den **u(q(d)) = q(u'(d))** gilt.



Änderbarkeit von Sichten

in SQL

- nur eine Basisrelation
- Schlüssel muss vorhanden sein
- keine Aggregatfunktionen, Gruppierung und Duplikateliminierung



Probleme mit Updates auf Views: Beispiel

```
CREATE VIEW WieHartAlsPrüfer (PersNr, Durchschnittsnote) AS
        SELECT PersNr, AVG(Note)
        FROM prüfen
        GROUP BY PersNr;


CREATE VIEW VorlesungenSicht AS
        SELECT v.Titel, v.SWS, p.Name
        FROM Vorlesungen v, Professoren p
        WHERE v.gelesenVon = p.PersNr;
INSERT INTO VorlesungenSicht
        VALUES ('Statistik', 4, 'Müller');
```

Materialized views (Snapshots)

Idee:

- Replikation der Daten in verteilten Umgebungen
- Caching der Ergebnisse von komplexen Queries (z.B. Data Warehouse) durch Materialisierung

- Bei Änderungen der Daten werden die Views aktualisiert
- Bei Updates der Views werden Änderungen zu den Relationen propagiert

Grobsyntax:

CREATE MATERIALIZED VIEW view-name [(column {, column …})]
[**FOR UPDATE**]
AS select-block

Zugriffskontrolle

Unterschiedliche Aspekte:

Data privacy: beschränkte Möglichkeiten, persönliche Daten zu speichern und zu verarbeiten (Datenschutz)

Data security: beschränkte Möglichkeiten, auf gespeicherte Daten zuzugreifen (Datensicherheit) – Zugriffskontrolle, Autorisierung

Maßnahmen zur Datensicherheit

- Organisationsmaßnahmen (z.B. Zugang zum Wireless-Netz)
- Technische Maßnahmen (Verschlüsselung)
- Maßnahmen des Betriebssystems (Firewalls, process isolation)
- Authentifizierung der DB-Benutzer
- Kontrolle der Zugriffsrechte der Benutzer beim Zugriff

Rechtevergabe in SQL: Prinzip

Subjekte haben *Rechte* (zur Ausführung von Operationen) auf *Objekten*.
Der Erzeuger einer Tabelle ist deren Eigentümer.
Rechte sind minimal, d.h. beschränkt auf den Eigentümer und Subjekte, denen explizit Rechte erteilt wurden.

Vergabe von Rechten mit der **GRANT**-Anweisung in SQL.
Grobsyntax:

GRANT { **ALL** | privilege {, privilege ...} } **ON** { table | view }
TO { **PUBLIC** | user {, user …} } [**WITH GRANT OPTION**]

Rechtevergabe in SQL: mögliche Rechte

SELECT	lesender Zugriff auf eine Relation
INSERT	Einfügen in eine Relation
UPDATE	Ändern von Tupeln einer Relation (ggf. nur bestimmte Attribute)
DELETE	Löschen von Tupeln einer Relation
CONNECT	Verbindung zum DBS aufnehmen ("Login"-Recht)
RESOURCE	Anlegen neuer Relationen (ggf. mit Limit für den Plattenplatz)
DBA	Datenbankadministration (z.B. Aufruf von Dienstprogrammen)
EXECUTE	Ausführung eines Anwendungsprogramms
IO_LIMIT	Beschränkung des Ressourcenverbrauchs für SQL-Anweisungen

.

.

.

Zugriffskontrolle – Beispiele (1)

Meier bekommt das Recht, Inhalte der Relation Studenten zu selektieren

GRANT SELECT ON Studenten **TO** Meier

Meier bekommt das Recht, Inhalte der Relation Studenten zu selektieren

REVOKE SELECT ON Studenten **FROM** Meier

Meier bekommt das Recht, Prüfungsergebnisse zu ändern

GRANT UPDATE ON prüfen **TO** Meier

Alle bekommen das Recht, die Raumbelegung anzusehen:

GRANT SELECT ON Raumbelegung **TO PUBLIC**

….

Zugriffskontrolle – Beispiele (2)

Meier bekommt das Recht, Personendaten der Studenten im 1..3 Semester zu lesen. Er kann das Recht an andere DB-User weitergeben:

CREATE VIEW YoungStudents **AS**
SELECT * **FROM** Students
WHERE Semester <= 3;

GRANT SELECT ON YoungStudents **TO** Meier
WITH GRANT OPTION;

Zugriffskontrolle – Beispiele (3)

Alice: Eigentümer von AliceTable.

Alice:
GRANT SELECT ON AliceTable **TO** Bob **WITH GRANT OPTION**;

Bob:
GRANT SELECT ON AliceTable **TO** Charlie **WITH GRANT OPTION**;

Alice:
REVOKE SELECT ON AliceTable **FROM** Bob;

Charlie:
GRANT SELECT ON AliceTable **TO** Bob; ← **NICHT möglich !**

REVOKE widerruft die Rechte von Bob und Charlie transitiv