Datenbanken

—

SQL zur Datendefinition und -manipulation

Thomas Studer

Institut für Informatik
Universität Bern

INSERT

Gegebenes Schema:

$$\texttt{Autos} = (\texttt{Marke}, \texttt{Farbe}, \texttt{Baujahr}, \texttt{FahrerId}) \ .$$

```
INSERT INTO Autos
  VALUES ('Audi', 'silber', 2008, 3)

INSERT INTO Autos
  VALUES ('Skoda', 'silber', 2009, Null)

INSERT INTO Autos (Baujahr, Farbe, Marke)
  VALUES (2009, 'silber', 'Skoda')
```

INSERT mehrere Tupel

```
INSERT INTO Autos
  VALUES ('Audi', 'silber', 2008, 3) ,
         ('Skoda', 'silber', 2009, Null)
```

fügt beide Tupel in die Relation `Autos` ein.

INSERT mit SQL Abfrage

```
INSERT INTO Autos
  SELECT 'Audi', 'silber', 2008, PersId
  FROM Personen
  WHERE Vorname = 'Eva' AND Nachname = 'Meier'
```

fügt das Tupel

$$(\texttt{'Audi'}, \texttt{'silber'}, 2008, 3)$$

in die Relation `Autos` ein, wenn `Eva Meier` die `PersId` 3 hat.

Sequenzen für Primärschlüssel

```
CREATE SEQUENCE PersIdSequence START 10

SELECT nextval('PersIdSequence')

INSERT INTO Personen
  VALUES ( nextval('PersIdSequence'),
           'Bob',
           'Müller' )

DROP SEQUENCE PersIdSequence
```

Weshalb Sequenzen?

```
SELECT nextval('PersIdSequence')
```

vs.

```
SELECT max(PersId)+1
FROM Personen
```

Die `max(PersId)+1` Variante soll NIE verwendet werden:

1. Zwei parallele Prozesse können dieselbe Id für zwei verschiedene neue Tupel erhalten.
2. Wenn ein Tupel gelöscht wird, kann ein anderes, neues Tupel eine bereits vergebebe Id erhalten.

DELETE

Eine *Löschanweisung* in SQL hat folgende Form:

`DELETE FROM` <*Relation*>
`WHERE` <*Prädikat*>

Eine `DELETE` Anweisung wird in zwei Schritten ausgeführt:

1. Markiere zuerst alle Tupel in <*Relation*>, auf die <*Prädikat*> zutrifft.
2. Lösche die markierten Tupel aus der Relation <*Relation*>.

```
DELETE FROM Autos
WHERE FahrerId IN
    ( SELECT PersId
      FROM Personen
      WHERE Vorname = 'Eva' AND Nachname = 'Meier' )
```

Beispiel für das Zwei-Schritt-Vorgehen

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim
Tim	Rob
Tom	Ted
Tom	Rik
Ted	Nik

```
DELETE FROM VaterSohn
WHERE Vater IN
    ( SELECT Sohn
      FROM VaterSohn )
```

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim

Mögliches (falsches) Ergebnis bei Ein-Schritt-Vorgehen

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim
Tim	Rob
Tom	Ted
Tom	Rik
Ted	Nik

```
DELETE FROM VaterSohn
WHERE Vater IN
    ( SELECT Sohn
      FROM VaterSohn )
```

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim
Ted	Nik

UPDATE

```
UPDATE Autos
SET Farbe = 'himmelblau'
WHERE Farbe = 'blau' AND Baujahr > 2010

UPDATE Autos
SET Baujahr = Baujahr - 1
```

Bemerkung: UPDATE Anweisungen werden ebenfalls in 2 Schritten ausgeführt (wie DELETE).

CREATE TABLE

```
CREATE TABLE TabellenName (
              Attribut_1 Domäne_1,
              ...
              Attribut_n Domäne_n )
```

Einige PostgreSQL Datentypen:

`integer`	Ganzzahlwert zwischen -2^{31} und $2^{31}-1$
`serial`	Ganzzahlwert mit Autoinkrement
`boolean`	Boolscher Wert
`char`	Einzelnes Zeichen
`char(`x`)`	Zeichenkette mit der Länge x, gegebenenfalls wird sie mit Leerzeichen aufgefüllt
`varchar(`x`)`	Zeichenkette mit *maximaler* Länge x
`text`	Zeichenkette mit beliebiger Länge

DEFAULT

```
CREATE TABLE Autos (
            Marke varchar(10),
            Farbe varchar(10) DEFAULT 'schwarz'  )
```

Damit fügt die Anweisung

```
INSERT INTO Autos (Marke) VALUES ('Audi')
```

das Tupel `('Audi', 'schwarz')` in die Tabelle `Autos` ein.

Nach Ausführen des Statements

```
INSERT INTO Autos VALUES ('VW', null)
```

hat die Tabelle `Autos` folgende Form:

Autos	
Marke	Farbe
Audi	schwarz
VW	-

SERIAL

```
CREATE TABLE Personen (
  PersId integer DEFAULT nextval('PersIdSequence'),
  Vorname varchar(10),
  Nachname varchar(10)   )
```

Wir können `Bob Müller` nun ganz einfach mit

```
INSERT INTO Personen (Vorname,Nachname)
  VALUES ('Bob','Müller')
```

Das Statement

```
CREATE TABLE TabellenName (
                    AttributName serial  )
```

ist äquivalent zu

```
CREATE SEQUENCE TabellenName_AttributName_seq;
CREATE TABLE TabellenName (
  AttributName integer DEFAULT
    nextval('TabellenName_AttributName_seq')
);
```

Constraints

```
CREATE TABLE TabellenName (
                Attribut_1 Domäne_1,
                ...
                Attribut_n Domäne_n,
                Constraint_1,
                ...
                Constraint_m )
```

Eine Ausnahme bilden `NOT NULL` Constraints, welche wir direkt hinter die entsprechenden Attribute schreiben.

Beispiel

```
CREATE TABLE Autos (
            Marke varchar(10),
            Farbe varchar(10),
            Baujahr integer NOT NULL,
            FahrerId integer,
            PRIMARY KEY (Marke, Farbe)  )
```

```
CREATE TABLE Personen (
            PersId integer PRIMARY KEY,
            Vorname varchar(10),
            Nachname varchar(10)  )
```

Fremdschlüssel

```
CREATE TABLE Autos (
  Marke varchar(10),
  Farbe varchar(10),
  Baujahr integer NOT NULL,
  FahrerId integer,
  PRIMARY KEY (Marke, Farbe),
  FOREIGN KEY (FahrerId) REFERENCES Personen  )
```

```
CREATE TABLE Autos (
            Marke varchar(10),
            Farbe varchar(10),
            Baujahr integer NOT NULL,
            FahrerId integer REFERENCES Personen ,
            PRIMARY KEY (Marke, Farbe)    )
```

UNIQUE

```
CREATE TABLE Personen (
              PersId integer PRIMARY KEY,
              Vorname varchar(10),
              Nachname varchar(10),
              UNIQUE (Vorname, Nachname)  )

CREATE TABLE Autos (
  Marke varchar(10),
  Farbe varchar(10),
  Baujahr integer NOT NULL,
  FahrerVorname varchar(10),
  FahrerNachname varchar(10),
  PRIMARY KEY (Marke, Farbe),
  FOREIGN KEY (FahrerVorname, FahrerNachname)
    REFERENCES Personen(Vorname, Nachname)    )
```

Fremdschlüssel

```
CREATE TABLE S (
   Id integer PRIMARY KEY )
```

```
CREATE TABLE R (
   Fr integer REFERENCES S )
```

S

Id
1
2

R

Fr
1
2

```
DELETE FROM S
WHERE Id = 1
```

```
UPDATE S
SET Id = 3
WHERE Id = 1
```

Fehlermeldung

update or delete on table “S” violates foreign key constraint.

Kaskadierung bei DELETE

```
CREATE TABLE R (
       Fr integer REFERENCES S ON DELETE CASCADE )
```

```
DELETE FROM S
WHERE Id = 1
```

R
Fr
2

S
Id
2

Kaskadierung bei UPDATE

```
CREATE TABLE R (
       Fr integer REFERENCES S ON UPDATE CASCADE )
```

```
UPDATE S
SET Id = 3
WHERE Id = 1
```

R
Fr
3
2

S
Id
3
2

Kaskade

```
CREATE TABLE S (
        Id integer PRIMARY KEY )

CREATE TABLE R (
        RId integer PRIMARY KEY,
        FrS integer REFERENCES S ON DELETE CASCADE)

CREATE TABLE Q (
        FrR integer REFERENCES R ON DELETE CASCADE)
```

Q

FrR
A
B

R

RId	FrS
A	1
B	2

S

Id
1
2

Kasade 2

Nach Ausführen der Anweisung

```
DELETE FROM S
WHERE Id = 1
```

erhalten wir folgende Instanzen:

Q
FrR
B

R	
RId	FrS
B	2

S
Id
2

SET NULL

```
CREATE TABLE S (
       Id integer PRIMARY KEY )

CREATE TABLE R (
       RId integer PRIMARY KEY,
       Fr integer REFERENCES S ON DELETE SET NULL
                               ON UPDATE SET NULL )
```

R

RId	Fr
A	1
B	2
C	3

S

Id
1
2
3

SET NULL 2

```
DELETE FROM S
WHERE Id = 1
```

R

RId	Fr
A	-
B	2
C	3

S

Id
2
3

```
UPDATE S
SET Id = 4
WHERE Id = 2
```

R

RId	Fr
A	-
B	-
C	3

S

Id
4
3

CHECK Constraints

```
CREATE TABLE Autos (
              Marke varchar(10),
              Farbe varchar(10),
              Baujahr integer,
              FahrerId integer,
              CHECK ( Farbe IN ('blau', 'rot')) )
```

```
CREATE TABLE Autos (
              Marke varchar(10),
              Farbe varchar(10),
              Mietbeginn integer,
              Mietende integer,
              FahrerId integer,
              CHECK (Mietbeginn < Mietende) )
```

Beispiel

Anforderungen:

1. Eine Datenbank soll Autos und ihre Automarken verwalten.
2. Zusätzlich werden noch Mechaniker verwaltet.

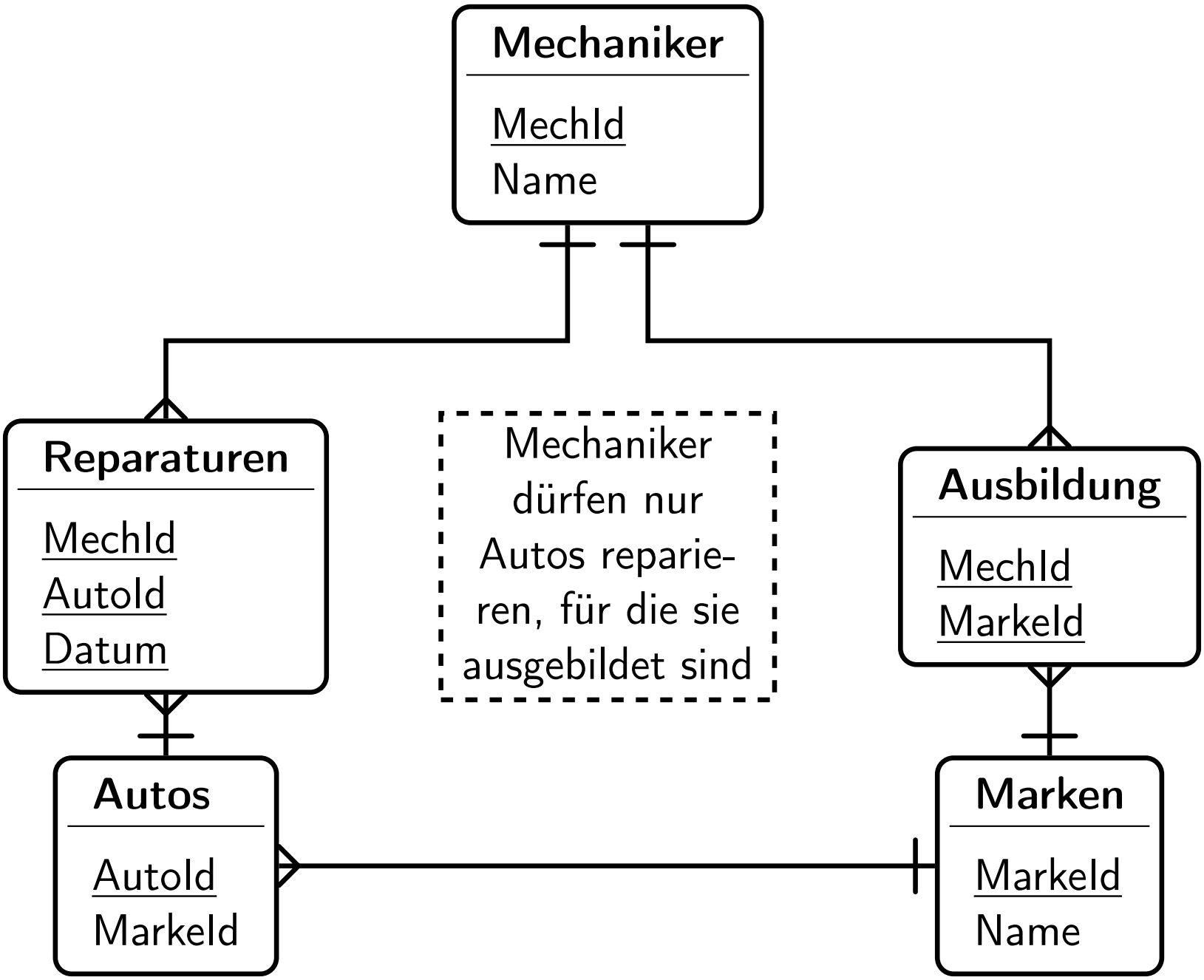
Es wird abgespeichert,

1. welche Mechaniker für welche Autos ausgebildet sind und
2. welche Mechaniker wann welche Autos repariert haben.

Einschränkungen:

1. Ein Mechaniker kann ein Auto mehrfach (an verschiedenen Daten) reparieren.
2. Er darf aber nur Autos reparieren, für deren Marken er ausgebildet wurde.

DB Schema


Mechaniker
MechId
Name
Reparaturen
MechId
AutoId
Datum
Mechaniker dürfen nur Autos reparieren, für die sie ausgebildet sind
Ausbildung
MechId
MarkeId
Autos
AutoId
MarkeId
Marken
MarkeId
Name

Tabellen

```
CREATE TABLE Mechaniker (
            MechId integer PRIMARY KEY,
            Name varchar(10) );

CREATE TABLE Marken (
            MarkeId integer PRIMARY KEY,
            Name varchar(10) );

CREATE TABLE Autos (
            AutoId integer PRIMARY KEY,
            MarkeId integer NOT NULL REFERENCES Marken );

CREATE TABLE Ausbildung (
            MechId integer REFERENCES Mechaniker,
            MarkeId integer REFERENCES Marken,
            PRIMARY KEY (MechId, MarkeId) );
```

Tabellen 2

```
CREATE TABLE Reparaturen (
                MechId integer REFERENCES Mechaniker,
                AutoId integer REFERENCES Autos,
                Datum integer,
                PRIMARY KEY (MechId, AutoId, Datum) )
```

Gerne würden wir auch folgenden Constraint hinzufügen:

```
CHECK (
   EXISTS (
      SELECT *
      FROM (Autos INNER JOIN Ausbildung
              USING (MarkeId)) AS Tmp
      WHERE Tmp.MechId = Reparaturen.MechId AND
            Tmp.AutoId = Reparaturen.AutoId ) )
```

Jedoch unterstützt PostgreSQL keine Subqueries in `CHECK` Constraints.

ALTER TABLE

```
ALTER TABLE Marken RENAME TO Automarken

ALTER TABLE Automarken RENAME Name TO Bezeichnung

ALTER TABLE Autos ADD Baujahr INTEGER

ALTER TABLE Automarken ADD UNIQUE(Bezeichnung)

ALTER TABLE Autos ADD CHECK(Baujahr > 2010)
```

Andere Syntax für NOT NULL Constraints, da diese nur ein Attribut betreffen:

```
ALTER TABLE Mechaniker ALTER Name SET NOT NULL
```

Views

Sichten (oder *Views*) sind virtuelle Tabellen, die mittels SQL Abfragen definiert werden.
Eine Sicht kann einen Ausschnitt aus einer Tabelle präsentieren, aber auch einen Verbund von mehreren Tabellen. Somit können Views:

1. komplexe Daten so strukturieren, dass sie einfach lesbar sind,
2. den Zugriff auf Daten einschränken, so dass nur ein Teil der Daten lesbar ist (anstelle von ganzen Tabellen),
3. Daten aus mehreren Tabellen zusammenfassen, um Reports zu erzeugen.

CREATE VIEW

```
CREATE VIEW AlteAutos AS
  SELECT *
  FROM Autos
  WHERE Baujahr <= ALL (
    SELECT Baujahr
    FROM Autos)
```

```
SELECT Vorname, Nachname
FROM Personen INNER JOIN AlteAutos ON (PersId = FahrerId)
```

Materialisierte Views

Materialisierte Views sind Views, die nicht bei jeder Verwendung neu berechnet werden, sondern nur einmal zum Zeitpunkt ihrer Kreation.

Dies bedeutet natürlich einen Performancegewinn bei Abfragen, da auf das vorberechnete Resultat der View zurückgegriffen werden kann.

Dafür können materialisierte Views temporär nicht-aktuelle Daten enthalten.

CREATE MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW AlteAutos AS
  SELECT *
  FROM Autos
  WHERE Baujahr <= ALL (
    SELECT Baujahr
    FROM Autos)
```

```
INSERT INTO Autos VALUES ('Audi', 'silber', 2008, 3)
```

```
REFRESH MATERIALIZED VIEW AlteAutos
```