SQL zur Datendefinition und Datenmanipulation

6

In diesem Kapitel zeigen wir, wie mit SQL Daten eingefügt, geändert und gelöscht werden können. Dabei studieren wir auch sogenannte Sequenzen. Das sind Datenbankobjekte mit denen eindeutige Werte erzeugt werden können. Diese sind wichtig, um Primärschlüsselattribute korrekt abzufüllen.

Weiter betrachten wir die Möglichkeiten von SQL zur Definition von Tabellen und Constraints. Dazu führen wir die wichtigsten Datentypen von PostgreSQL ein und zeigen, wie Defaultwerte für Attribute definiert werden können. Ausserdem untersuchen wir die Funktionsweise von unique, not null, primary key, foreign key und check Constraints und geben Beispiele, wie Änderungen an Attributen eine Kaskade von Änderungen in anderen Tabellen zur Folge haben können. Zum Schluss führen wir noch dynamische und materialisierte Views ein.

6.1 Datenmanipulation

Bisher haben wir SQL als Sprache zur Abfrage einer Datenbank betrachtet. Nun wenden wir uns dem Problem zu, mit Hilfe von SQL Daten hinzuzufügen, zu entfernen und zu modifizieren. Ausserdem wollen wir SQL zur Definition von Datenbankschemata verwenden.

Um Daten in eine Relation *einzufügen*, spezifizieren wir ein Tupel, das eingefügt werden soll, oder schreiben eine Abfrage, deren Ergebnis eine Menge von Tupeln ist, die eingefügt werden sollen. Dabei ist zu beachten, dass die Attributwerte der einzufügenden Tupel zu den Domänen der betroffenen Attribute gehören müssen. Ausserdem müssen die einzufügenden Tupel die korrekte Stelligkeit besitzen.

Wir betrachten wieder die Relation `Autos` über dem Schema

```
(Marke, Farbe, Baujahr, FahrerId).
```


T. Studer, *Relationale Datenbanken*,
https://doi.org/10.1007/978-3-662-58976-2_6

Wie bereits beschrieben, können wir mit der Anweisung

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

ein neues Tupel in die Relation `Autos` einfügen.

Es können auch explizit `Null` Werte eingefügt werden, bspw. falls der Fahrer unbekannt ist. Die entsprechende Anweisung lautet dann:

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

Äquivalent dazu ist folgende Anweisung:

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

Hierbei geben wir die Attribute des einzufügenden Tupels explizit an. Diese dürfen dann in einer beliebigen Reihenfolge auftreten. Attribute, welche nicht angegeben werden, erhalten den Wert `Null`.

Die einzufügenden Tupel können auch mit Hilfe einer SQL Abfrage erzeugt werden. Die Anweisung

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

fügt das Tupel

```
('Audi', 'silber', 2008, 3)
```

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

Ein Problem bei der Formulierung von `INSERT` Statements besteht darin, einen eindeutigen Wert für die Primärschlüssel-Attribute zu finden. Wir betrachten die Relation `Personen` über dem Schema

```
(PersId, Vorname, Nachname),
```

wobei das Attribut `PersId` als Primärschlüssel dient. Wenn wir nun ein neues Tupel in diese Relation einfügen wollen, so müssen wir einen neuen, eindeutigen Wert für das Attribut `PersId` einfüllen.

Einen solchen Wert können wir mit einer sogenannten *Sequenz* kreieren. Eine Sequenz ist ein Datenbank-Objekt, welches eine Folge von fortlaufenden Nummern erzeugen kann. Wir erstellen eine Sequenz mit der Anweisung

```
CREATE SEQUENCE PersIdSequence START 10
```

Hier ist `PersIdSequence` der Name der Sequenz und 10 ihr Startwert. Mit der Anweisung

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

können wir die nächste Zahl der Sequenz abfragen. Hier erhalten wir die Antwort 10. Der Wert der Sequenz wird mit der Funktion `nextval` automatisch um eins erhöht. Wenn wir also die Abfrage

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

nochmals ausführen, so erhalten wir nicht mehr den Wert 10, sondern den Wert 11. Wir können diese Sequenz in einer `INSERT` Anweisung wie folgt verwenden:

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

Wird die Sequenz nicht mehr benötigt, so kann sie mit

```
DROP SEQUENCE PersIdSequence
```

gelöscht werden.
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*>.

Wird keine `WHERE` Klausel angegeben, so werden alle Tupel aus <*Relation*> gelöscht. Das Prädikat in der `WHERE` Klausel kann so komplex sein, wie in `SELECT` Anweisungen. Eine Löschanweisung bezieht sich immer nur auf eine Relation. Sollen Daten aus mehreren Relationen gelöscht werden, so muss eine `DELETE` Anweisung für jede Relation ausgeführt werden.

Wollen wir alle Autos löschen, die einer Person mit Namen `Eva Meier` gehören, so verwenden wir folgende Anweisung:

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

Es ist wesentlich, dass die Löschoperation zuerst die zu löschenden Tupel markiert und diese erst anschliessend aus der Relation entfernt. Falls die Tupel im ersten Schritt nicht nur markiert sondern direkt gelöscht würden, so wäre das Resultat der Löschoperation von der Reihenfolge der Tupel abhängig.

Zur Illustration betrachten wir nochmals die Tabelle `VaterSohn` aus Beispiel 5.4.

VaterSohn

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

Wir führen nun folgende Löschanweisung aus:

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

Mit dem Zwei-Schritt-Verfahren der `DELETE` Anweisung (zuerst markieren, dann löschen) erhalten wir folgende Tabelle:

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim

Würden wir hingegen die Tabelle `VaterSohn` von oben nach unten durchgehen und diejenigen Tupel, auf die das Prädikat

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

zutrifft, sofort löschen, dann würden wir folgendes (falsches) Resultat erhalten:

VaterSohn

Vater	Sohn
Bob	Tom
Bob	Tim
Ted	Nik

Neben Einfüge- und Löschoperationen unterstützt SQL auch *Änderungsanweisungen*, mit denen der Wert von einzelnen Attributen geändert werden kann. Die `UPDATE` Operation funktioniert analog zu den bereits eingeführten `INSERT` und `DELETE` Anweisungen. Wir betrachten hier nur zwei einfache Beispiele.

Nehmen wir an, wir wollen bei allen Autos, welche nach 2010 gebaut wurden, die Farbe `blau` zu `himmelblau` ändern. Dies wird durch folgende Anweisung erreicht.

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

Es ist auch möglich den neuen Attributwert aus dem alten Wert zu berechnen. Folgendes Beispiel macht alle Autos ein Jahr älter.

```
UPDATE Autos
SET Baujahr = Baujahr - 1
```

6.2 Erstellen von Tabellen

Wie bereits früher beschrieben, können wir mit einer `CREATE TABLE` Anweisung eine leere Tabelle erzeugen. Diese Anweisung hat folgende Form:

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

Wir geben also den Namen der Tabelle und ihre Attribute mit den entsprechenden Domänen an. Für diese Domänen stehen uns verschiedene in PostgreSQL vordefinierte Datentypen zur Verfügung. Die folgende Liste gibt die für uns wichtigsten Datentypen an:

`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

Der Datentyp `varchar(`x`)` gehört zum SQL-Standard, wohingegen `text` ein PostgreSQL spezifischer Datentyp ist. Laut der PostgreSQL Dokumentation gibt es zwischen diesen beiden Datentypen keine Performance-Unterschiede. Der Datentyp `serial` wird im nächsten Abschnitt noch genauer besprochen.

Darüber hinaus unterstützt PostgreSQL die gängigen SQL Datentypen zur Speicherung von Datums- und Uhrzeitwerten, Gleit- und Fliesskommazahlen, sowie grossen Objekten (LOBs). Auch anwendungsspezifische Datentypen für bspw. geometrische Objekte oder Netzwerkadressen sind in PostgreSQL vordefiniert.

6.3 Default Werte

In einem `CREATE TABLE` Statement können wir Default Werte für Attribute angeben. Mit folgender Anweisung erzeugen wir eine Tabelle `Autos` bei der das Attribut `Farbe` den Default Wert `schwarz` hat.

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

Das heisst, falls wir ein neues Tupel in dieser Tabelle erzeugen und dem Attribut `Farbe` keinen Wert zuweisen, so wird dieses Attribut auf den Wert `schwarz` gesetzt. Somit fügt die Anweisung

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

das Tupel

```
('Audi', 'schwarz')
```

in die Tabelle `Autos` ein.

Trotz des spezifizierten Default Wertes ist es möglich das Attribut `Farbe` auf `Null` zu setzen. Nach Ausführen des Statements

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

hat die Tabelle `Autos` folgende Form:

Autos

Marke	Farbe
Audi	schwarz
VW	-

Ein Default Wert kann nicht nur durch eine Konstante angegeben werden. Er kann auch durch einen Ausdruck spezifiziert werden, der berechnet wird, wenn ein Tupel in die Tabelle eingefügt wird. Das übliche Beispiel dazu ist das Auslesen einer Sequenz. Betrachten wir noch einmal die Relation `Personen` und die Sequenz `PersIdSequence` aus Abschn. 6.1. Wir erzeugen die Tabelle `Personen` nun mit

```
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')
```

in diese Tabelle einfügen und brauchen uns nicht um das Erzeugen der eindeutigen `PersId` zu kümmern.

Diese Konstellation tritt so häufig auf, dass es dafür eine Abkürzung gibt, nämlich den Datentypen `serial`. Dieser ist nicht ein eigentlicher Datentyp, sondern nur eine notationelle Annehmlichkeit, um eindeutige Attribute zu spezifizieren. 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')
);
```

Anmerkung 6.1. Diese Darstellung des Typs `serial` ist ein wenig zu einfach. Es wird z. B. zusätzlich sichergestellt, dass beim Löschen der Tabelle auch die Sequenz gelöscht wird. Eine genaue Beschreibung findet man in der PostgreSQL Documentation unter https://www.postgresql.org/docs/current/static/datatype-numeric.html

6.4 Constraints

Im Kapitel zum Relationenmodell haben wir verschiedene Constraints betrachtet, welche wir von Tabellen verlangen können. Nun geht es darum, wie wir diese in SQL formulieren können. Ganz allgemein ist der Aufbau einer `CREATE TABLE` Anweisung mit Constraints der folgende:

```
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.

Wir nehmen nun an, wir wollen die Tabelle `Autos` erzeugen, mit dem Primärschlüssel (`Marke`, `Farbe`) und einem `NOT NULL` Constraint auf dem Attribut `Baujahr`. Dann verwenden wir folgende Anweisung:

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

Falls der Primärschlüssel aus nur *einem* Attribut besteht, so können wir das Schlüsselwort `PRIMARY KEY` auch hinter das entsprechende Attribut schreiben. Wir können also die Tabelle `Personen` folgendermassen erzeugen:

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

Es fehlt nun noch die Fremdschlüsselbeziehung zwischen diesen Tabellen. Diese können wir mit dem Schlüsselwort `FOREIGN KEY` angeben, wenn wir die Tabelle `Autos` erzeugen.

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

Da der Fremdschlüssel aus nur einem Attribut besteht, können wir auch hier eine Notation verwenden, bei welcher der Constraint direkt hinter dem Attribut angegeben wird:

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

Mit Hilfe eines UNIQUE Constraints können wir verlangen, dass in der Tabelle Personen jede Kombination von Vorname und Nachname nur einmal vorkommt. Die entsprechende CREATE TABLE Anweisung lautet dann:

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

Es ist nun auch möglich in der Tabelle Autos ein Tupel der Tabelle Personen via die Attribute Vorname und Nachname zu referenzieren. Dazu müssen wir nach dem Schlüsselwort REFERENCES nicht nur die Tabelle, sondern auch noch die eindeutigen Attribute angeben. Falls wir diese Attribute nicht angeben (wie im Beispiel weiter oben), so wird automatisch der Primärschlüssel referenziert. Eine CREATE TABLE Anweisung mit einer Referenz auf UNIQUE Attribute wird folgendermassen formuliert:

```
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)    )
```

6.5 Kaskadierung

In diesem Abschnitt diskutieren wir, wie sich Änderungen in einer Tabelle via Fremdschlüsselbeziehung auf eine andere Tabelle auswirken können. Wir betrachten dazu das folgende einfache Beispiel. Wir erzeugen zwei Tabellen durch:

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

und

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

Das Attribut Fr in der Tabelle R ist also ein Fremdschlüssel auf die Tabelle S.

Wir nehmen an wir haben folgenden Zustand in den beiden Tabellen:

R
Fr
1
2

S
Id
1
2

Es ist nun nicht möglich einen Eintrag in der Tabelle `S` zu löschen oder zu ändern, da dies die referentielle Integrität verletzen würde. Das heisst, die Anweisungen

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

und

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

können nicht ausgeführt werden und liefern die Fehlermeldung:

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

Im `FOREIGN KEY` Constraint können wir jedoch auch spezifizieren, dass Änderungen propagiert werden sollen. Man sagt dazu auch, Änderungen werden *kaskadiert*. Wir können dieses Verhalten für `DELETE` und `UPDATE` Operationen getrennt angeben.

Wir nehmen an, die Tabelle `R` wurde erzeugt mit:

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

Dann führt die Löschanweisung

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

zu folgenden Tabellen:

R
Fr
2

S
Id
2

Es wird also die Löschoperation in `S` ausgeführt. Um die referentielle Integrität zu erhalten, wird die Löschung in der Tabelle `R` propagiert und führt dort zur Löschung des Eintrags mit dem Fremdschlüssel `1`.

Für Änderungsoperationen funktioniert das analog. Wir nehmen an, die Tabelle `R` wurde erzeugt mit:

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

Dann führt die Anweisung

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

zu folgenden Tabellen:

R
Fr
3
2

S
Id
3
2

Wiederum wird die Änderung in `S` ausgeführt und nach `R` propagiert.

Das Schlüsselwort `CASCADE` deutet an, dass eine Änderungsanweisung eine ganze Reihe von Tabellen betreffen kann, falls diese durch eine Kette von Fremdschlüsseln verbunden sind. Wir betrachten nun drei Tabellen:

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

und

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

sowie

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

Wir nehmen an wir haben folgenden Instanzen:

Q
FrR
A
B

R	
RId	**FrS**
A	1
B	2

S
Id
1
2

Eine Löschanweisung in der Tabelle `S` führt nun zu einer Löschoperation in allen drei Tabellen. Nach Ausführen der Anweisung

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

erhalten wir nämlich folgende Instanzen:

Q
FrR
B

R	
RId	FrS
B	2

S
Id
2

Neben dem Kaskadieren von Änderungen gibt es auch noch die Möglichkeit, Fremdschlüsselattribute automatisch auf `Null` zu setzen, um die referentielle Integrität zu gewährleisten. Dazu erzeugen wir den Fremdschlüssel mit dem Zusatz `SET NULL`. In folgendem Beispiel wollen wir dies sowohl bei `DELETE` wie auch bei `UPDATE` Operationen verlangen. Das geht wie folgt:

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

und

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

Wir nehmen folgende Instanzen an:

R	
RId	Fr
A	1
B	2
C	3

S
Id
1
2
3

Die Löschanweisung

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

führt nun zu folgendem Resultat:

R	
RId	Fr
A	-
B	2
C	3

S
Id
2
3

Wenn wir jetzt noch die Änderungsanweisung

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

ausführen, so erhalten wir

R	
RId	**Fr**
A	-
B	-
C	3

S
Id
4
3

Zusätzlich zu den bisher beschriebenen Constraints bietet PostgreSQL die Möglichkeit sogenannte `CHECK` Constraints zu definieren. Das sind allgemeine Bedingungen, welche bei jeder Datenmanipulation überprüft werden.

Mit folgender `CREATE TABLE` Anweisung können wir verlangen, dass das Attribut `Farbe` nur die Werte `blau` und `rot` annehmen darf.

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

Mit einem `CHECK` Constraint können wir auch verlangen, dass ein Beginndatum vor einem Enddatum liegen muss. Beispielsweise wenn die Vermietung von Autos mit folgender Tabelle verwaltet wird:

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

Zum Schluss betrachten wir ein DB-Schema für folgende Anforderungen. Eine Datenbank soll Autos und ihre Automarken verwalten. Zusätzlich werden noch Mechaniker verwaltet. Es wird abgespeichert, welche Mechaniker für welche Autos ausgebildet sind. Weiter wird abgespeichert welche Mechaniker wann welche Autos repariert haben. Ein Mechaniker kann ein Auto mehrfach (an verschiedenen Daten) reparieren. Er darf aber nur Autos reparieren, für deren Marken er ausgebildet wurde. Wir erhalten das DB-Schema aus Abb. 6.1.

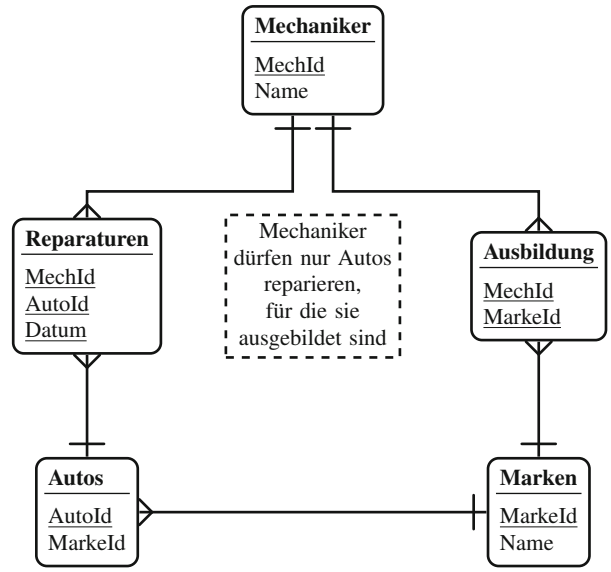

Abb. 6.1 DB-Schema für Autoreparaturen

Wir erzeugen dieses Schema mit den folgenden Anweisungen.

```
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) );

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

Es fehlt noch die Bedingung, dass Mechaniker nur Reparaturen an Marken ausführen, für die sie ausgebildet sind. Gerne würden wir auch folgenden Constraint zu der Anweisung `CREATE TABLE Reparaturen` 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. Der Grund dafür ist, dass die Überprüfung von solchen Constraints nur *global* auf der ganzen Datenbank erfolgen kann und somit äusserst aufwändig ist. Im obigen Beispiel kann eine `DELETE` Anweisung auf der Tabelle `Ausbildung` dazu führen, dass die Bedingung auf der Tabelle `Reparaturen` verletzt wird.

6.6 Ändern von Tabellen

PostgreSQL bietet eine Reihe von Operationen an, um die Tabellenstruktur zu ändern. Wir können Tabellen *umbenennen* mit der Anweisung:

`ALTER TABLE` <*Tabelle*> `RENAME TO` <*Name*>

Wir können die Tabelle `Marken` zu `Automarken` umbenennen mit:

```
ALTER TABLE Marken RENAME TO Automarken
```

Attribute können ebenfalls umbenannt werden. Dazu verwenden wir:

`ALTER TABLE` <*Tabelle*> `RENAME` <*Attribut*> `TO` <*Name*>

Wir ändern also das Attribut `Name` in `Automarken` mittels der Anweisung:

```
ALTER TABLE Automarken RENAME Name TO Bezeichnung
```

Wir können Attribute *hinzufügen* mit

`ALTER TABLE` <*Tabelle*> `ADD` <*Attributname*> <*Datentyp*>

Wir fügen also das Attribut `Baujahr` zu `Autos` hinzu durch:

```
ALTER TABLE Autos ADD Baujahr INTEGER
```

Ausserdem ist es möglich Constraints hinzuzufügen mit

`ALTER TABLE` <*Tabelle*> `ADD` <*Constraint*>

Es macht Sinn zu verlangen, dass die Werte des Attributs `Bezeichnung` in der Tabelle `Automarken` eindeutig sind. Wir erreichen das durch:

```
ALTER TABLE Automarken ADD UNIQUE(Bezeichnung)
```

Das geht auch mit anderen Constraints. Beispielsweise können wir verlangen, dass keine alten Autos abgespeichert werden sollen:

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

Wir können auch `NOT NULL` Constraints hinzufügen. Da diese nur eine einzelne Spalte betreffen, hat die entsprechende Anweisung eine etwas andere Syntax. Wir verlangen nun, dass jeder Mechaniker einen Namen haben muss:

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

Falls eine Tabelle `A` einen Fremdschlüssel auf eine Tabelle `B` besitzt und umgekehrt `B` auch einen Fremdschlüssel auf `A` besitzt, so muss mindestens einer dieser Constraints mit einer `ALTER TABLE` Anweisung hinzugefügt werden. Dies wird in folgendem Beispiel illustriert.

Beispiel 6.2. Wir betrachten wiederum `Autos` und `Personen`. Jetzt nehmen wir an, jede Person kann höchstens *ein* Auto fahren und jedes Auto kann von höchstens *einer* Person gefahren werden. Wir wollen also sozusagen Autos-Personen Paare bilden, wobei diese Paare auch ändern können, wenn eine Person ihr Auto wechselt. Dazu fügen wir beiden Tabellen ein neues Attribut `PaarId` hinzu. Wir können die Tabellen `Autos` und `Personen` wie folgt erzeugen.

```
CREATE TABLE Autos (
       AutoId integer PRIMARY KEY,
       PaarId integer UNIQUE );

CREATE TABLE Personen (
       PersId integer PRIMARY KEY,
       PaarId integer UNIQUE
         REFERENCES Autos(PaarId) DEFERRABLE );

ALTER TABLE Autos ADD
  FOREIGN KEY (PaarId)
    REFERENCES Personen(PaarId) DEFERRABLE
```

Die Fremdschlüssel-Beziehung von `Autos` zu `Personen` kann nicht bereits bei der Erzeugung der Tabelle `Autos` verlangt werden, da es zu diesem Zeitpunkt die Tabelle `Personen` noch gar nicht gibt. Das Schlüsselwort `DEFERRABLE` sagt, dass die Überprüfung des references Constraints aufschiebbar ist, mehr dazu später in Beispiel 8.2.

Natürlich können Attribute und Constraints nicht nur hinzugefügt werden, Es ist auch möglich diese zu entfernen. Die Anweisung

```
ALTER TABLE Autos DROP Baujahr
```

beispielsweise entfernt das Attribut `Baujahr` aus der Tabelle `Autos`. Automatisch werden damit auch allfällige Constraints auf der Tabelle `Autos`, welche das Attribut `Baujahr` betreffen, entfernt.

6.7 Views

Sichten (oder *Views*) in einem Datenbanksystem sind virtuelle Tabellen, die mittels SQL Abfragen definiert werden. Virtuell heisst dabei, dass es sich zwar nicht um „echte“ Tabellen handelt, dass sie aber in Abfragen wie Tabellen verwendet werden können.

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.

Um eine View zu kreieren, verwenden wir die Anweisung `CREATE VIEW`. Wir müssen dann der View einen Namen geben und mit Hilfe einer Query spezifizieren, wie die View berechnet werden soll. Die vollständige Anweisung lautet somit

`CREATE VIEW` <*Name*> `AS` <*Query Expression*>,

wobei <*Query Expression*> eine legale SQL Abfrage sein muss und <*Name*> die Bezeichnung der View ist.

Beispiel 6.3. Wir betrachten erneut die Tabellen `Autos` und `Personen` aus Beispiel 2.5:

Autos

Marke	**Farbe**	**Baujahr**	**FahrerId**
Opel	silber	2010	1
Opel	schwarz	2010	2
VW	rot	2014	2
Audi	schwarz	2014	3

Personen

PersId	Vorname	Nachname
1	Tom	Studer
2	Eva	Studer
3	Eva	Meier

Mit folgendem Statement erzeugen wir eine View, welche die ältesten Autos enthält.

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

Wir können diese View dann verwenden, um die Fahrer der ältesten Autos zu finden. Dazu formulieren wir folgende Abfrage:

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

Diese liefert das Resultat:

Vorname	Nachname
Tom	Studer
Eva	Studer

Zusammenfassend soll festgehalten werden, dass auf einer View die üblichen Abfragen anwendbar sind. Dabei wird bei jeder Abfrage die View dynamisch neu berechnet. Views können auch über andere Views definiert werden. Rekursive Views werden jedoch nicht unterstützt.

Ebenso wie Relationen sind Views persistente Objekte. Ihre Lebensdauer wird durch den Befehl `DROP VIEW` beendet. Durch

```
DROP VIEW AlteAutos
```

wird die View im obigen Beispiel gelöscht. Die Abfrage nach den Fahrern der ältesten Autos wird damit ungültig.

Neben den Views wie wir sie oben beschrieben haben, unterstützt PostgreSQL auch *materialisierte Views*. Das sind Views, die nicht bei jeder Verwendung neu berechnet werden, sondern nur einmal zum Zeitpunkt ihrer Erzeugung. 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. Falls diejenigen Daten, auf denen die View basiert, geändert wurden, so muss auch die View neu berechnet werden.

Wir können `AlteAutos` wie folgt als materialisierte View erzeugen:

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

Die Abfrage aus Beispiel 6.3 nach den Fahrern der ältesten Autos liefert wiederum das Resultat:

Vorname	Nachname
Tom	Studer
Eva	Studer

Wir fügen nun ein Auto mit Baujahr 2008 hinzu:

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

Da wir nun eine materialisierte View verwenden, erhalten wir mit der Abfrage nach den Fahrern der ältesten Autos immer noch das Resultat: `Tom Studer` und `Eva Studer`. Dieses falsche Ergebnis erhalten wir, weil zwar die Grunddaten aktualisiert wurden, die View aber noch auf einem alten (inzwischen falschen) Stand ist.

Um das richtige Resultat zu erhalten, müssen wir die View aktualisieren. Dies geschieht mit der Anweisung:

```
REFRESH MATERIALIZED VIEW AlteAutos
```

Damit wird die View neu berechnet und enthält wieder die korrekten Daten. Die Abfrage nach den Fahrern der ältesten Autos liefert jetzt das richtige Resultat:

Vorname	Nachname
Eva	Meier

Zum Schluss können wir die materialisierte View wieder löschen mit der Anweisung:

```
DROP MATERIALIZED VIEW AlteAutos
```

Weiterführende Literatur[1]

1. ANSI: Database language SQL (2011). Dokument X3.135-2011
2. The PostgreSQL Global Development Group: Postgresql documentation, data types (2018). https://www.postgresql.org/docs/current/static/datatype.html. Zugegriffen am 11.06.2019

[1] Die allgmeine Literatur zu SQL haben wir im vorangehenden Kapitel angegeben. Hier wiederholen wir nur die Referenz auf den SQL Standard [1]. Für eine vollständige Beschreibung aller Datentypen, welche PostgreSQL unterstützt, verweisen wir auf die online verfügbare PostgreSQL Dokumentation [2]. Dort finden sich unter anderem auch genaue Angaben zum benötigen Speicherplatz der verschiedenen Datentypen.