SQL Abfragen

5

SQL (Structured Query Language) ist die wichtigste Sprache im Bereich der relationalen Datenbanksysteme. Obwohl sie als *Query Language* bezeichnet wird, bietet SQL weit mehr Möglichkeiten als blosse Datenabfrage. So kann man etwa die Struktur von Daten definieren, Daten in einer Datenbank modifizieren und Sicherheitsbedingungen aufbauen. Nach einer Kurzeinführung in SQL werden wir in diesem Kapitel im Detail zeigen, wie mit SQL Datenbankabfragen formuliert werden können. Dabei betrachten wir auch die verschiedenen Arten von Join-Operationen, Gruppierung und Aggregation, sowie rekursive Queries.

5.1 SQL ganz kurz

In einer relationalen Datenbank werden sämtliche Daten in Tabellen gespeichert. In einem ersten Schritt müssen wir also eine solche Tabelle erzeugen. Dies geschieht mit einem `CREATE TABLE` Statement, welches eine leere Instanz (d. h. eine Relation, welche keine Tupel enthält) über einem Schema erstellt. Eine einfache Form des `CREATE TABLE` Statements ist:

```
CREATE TABLE TabellenName (
              Attribut_1 Domäne_1,
              Attribut_2 Domäne_2,
              Attribut_3 Domäne_3 )
```

Damit wird eine leere Instanz über dem Schema

```
(Attribut_1 : Domäne_1, Attribut_2 : Domäne_2,
 Attribut_3 : Domäne_3 )
```


T. Studer, *Relationale Datenbanken*,
https://doi.org/10.1007/978-3-662-58976-2_5

erzeugt. Dieser Instanz wird der Name `TabellenName` gegeben.

Mit folgendem Statement erzeugen wir eine Tabelle `Personen` mit den Attributen `PersNr`, `Name` und `GebDatum`. Die Domäne `INTEGER` bezeichnet dabei die ganzen Zahlen und `VARCHAR(30)` meint Zeichenketten mit maximal 30 Zeichen.

```
CREATE TABLE Personen (
   PersNr INTEGER,
   Name VARCHAR(30),
   GebDatum INTEGER)
```

Nun können wir Daten in die erzeugte Tabelle einfüllen. Dies erfolgt mit Hilfe eines `INSERT` Statements, welches folgende Form hat:

```
INSERT INTO TabellenName
  VALUES (Wert_1, Wert_2, Wert_3);
```

Dabei wird das Tupel

```
(Wert_1, Wert_2, Wert_3)
```

in die Tabelle `TabellenName` eingefügt. Konkret könnte das wie folgt aussehen. Die Zeichenkette geben wir in *einfachen Anführungsstrichen* an.

```
INSERT INTO  Personen VALUES (1, 'Tom', 19720404)
```

Wir können auch `Null` Werte einfügen, indem wir an der entsprechenden Stelle im Tupel `null` angeben. So zum Beispiel, falls das Geburtsdatum unbekannt ist:

```
INSERT INTO  Personen VALUES (2, 'Eva', null)
```

Mit einem `SELECT` Statement können wir die Daten einer Tabelle wieder auslesen. Die einfachste Form eines solchen Statements ist:

```
SELECT * FROM TabellenName
```

Damit wird der Inhalt der Tabelle `TabellenName` ausgegeben. Somit können wir unsere Personendaten auslesen mit:

```
SELECT * FROM Personen
```

Dieses Statement liefert folgende Tabelle (nach dem Ausführen der obigen beiden `INSERT` Statements):

PersNr	Name	GebDatum
1	Tom	19720404
2	Eva	Null

Anmerkung 5.1. An dieser Stelle ist es wichtig, dass SQL Gross-/Kleinschreibung nicht unterscheidet. Das heisst, PostgreSQL setzt alle Identifier (Namen von Tabellen,

Attributen, etc.) automatisch in Kleinschreibung. Mit dem `CREATE TABLE` Statement weiter oben wird also intern eine Tabelle `personen` erzeugt, welche die Attribute `persnr`, `name` und `gebdatum` enthält.

Dieses Übersetzen in Kleinbuchstaben wird nicht nur beim Erzeugen der Tabelle angewendet, sondern bei allen Statements. Die obige Abfrage wird also übersetzt zu

```
SELECT * FROM personen
```

Damit passt sie wieder zu der intern verwendeten Kleinschreibung.

Das bedeutet auch, dass bei der Resultate-Tabelle in PostgreSQL die Attributnamen `persnr`, `name` und `gebdatum` ausgegeben werden. Wir werden das jedoch bei unseren Beispielen nicht berücksichtigen und Gross-/Kleinschreibung verwenden.

Zum Schluss wollen wir unsere Datenbank noch aufräumen. Mit folgendem `DROP TABLE` Statement löschen wir die Tabelle `TabellenName`.

```
DROP TABLE TabellenName
```

Für unsere `Personen` Tabelle heisst das also:

```
DROP TABLE Personen
```

5.2 Einfache Abfragen

Wir haben im vorherigen Abschnitt eine einfache Form des `SELECT` Statements gesehen. Die allgemeine Struktur eines `SELECT` Statements besteht aus den drei Klauseln `SELECT`, `FROM` und `WHERE`:

- Die `SELECT` Klausel entspricht der Projektionsoperation der relationalen Algebra. Sie wird verwendet, um diejenigen Attribute aufzulisten, die im Ergebnis einer Abfrage gewünscht werden.
- Die `FROM` Klausel entspricht der Bildung von kartesischen Produkten in der relationalen Algebra. Sie listet die Relationen auf, die bei der Evaluation eines Ausdrucks durchgesehen werden.
- Die `WHERE` Klausel entspricht dem Selektionsprädikat der relationalen Algebra. Sie besteht aus einem Prädikat, in dem Attribute derjenigen Relationen auftreten, die in der `FROM`-Klausel vorkommen.

Wir dürfen also die Selektionsoperation $\sigma_{\Theta}(R)$ der relationalen Algebra nicht mit der `SELECT` Klausel von SQL verwechseln. Eine typische SQL Abfrage hat die Form

```
SELECT A1,A2,...,Am
FROM R1,R2,...,Rn
WHERE Θ
```

Dabei repräsentiert jedes $\mathtt{A}_i$ ein Attribut ($1 \leq i \leq m$) und jedes $\mathtt{R}_i$ eine Relation ($1 \leq j \leq n$); Θ ist ein Prädikat. Diese Abfrage ist äquivalent zum Ausdruck

$$\pi_{\mathtt{A}_1,\ldots,\mathtt{A}_m}(\sigma_{\Theta}(\mathtt{R}_1 \times \cdots \times \mathtt{R}_n))$$

der relationalen Algebra. Wir nennen diesen Ausdruck die *kanonische Übersetzung* der SQL Abfrage. Wird bei der SQL Abfrage die `WHERE` Klausel weggelassen, so ist dies gleichbedeutend damit, dass wir für Θ das Prädikat `true` verwenden. Es ist ferner zu beachten, dass das Ergebnis einer SQL Abfrage mehrere Kopien desselben Tupels enthalten kann (im Gegensatz zu Ausdrücken der relationalen Algebra, welche immer Mengen beschreiben).

SQL bildet also das kartesische Produkt der Relationen, die in der `FROM` Klausel genannt sind, führt die Selektion der relationalen Algebra unter Benutzung des Prädikats der `WHERE` Klausel durch und projiziert das Ergebnis auf die Attribute der `SELECT` Klausel. Dabei arbeitet SQL mit Multimengen und nicht mit gewöhnlichen Mengen.

`SELECT` Klauseln

In diesem Abschnitt bezeichnet `Autos` die Relation `Autos` aus Beispiel 2.5, die über den Attributen `Marke`, `Farbe`, `Baujahr` und `FahrerId` eingeführt worden ist. Zur Erinnerung geben wir hier die Tabelle nochmals an:

Autos

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

Das Ergebnis einer SQL Abfrage können wir ebenfalls als Tabelle darstellen. Beispielsweise erhalten wir mit der SQL Abfrage

```
SELECT Marke, Baujahr
FROM Autos
```

die Tabelle

Marke	**Baujahr**
Opel	2010
Opel	2010
VW	2014
Audi	2014

Wie bereits gesagt, werden Duplikate nicht entfernt. Man kann jedoch die Elimination von Duplikaten erzwingen, indem man das Schlüsselwort `DISTINCT` einfügt. Die Abfrage

```
SELECT DISTINCT Marke, Baujahr
FROM Autos
```

antwortet

Marke	Baujahr
Opel	2010
VW	2014
Audi	2014

Die `SELECT` Klausel kann auch arithmetische Ausdrücke mit Konstanten und den Operatoren $+$, $-$, $*$ und / enthalten. Zum Beispiel liefert die SQL-Abfrage

```
SELECT DISTINCT Marke, Baujahr+1
FROM Autos
```

die Tabelle

Marke	Baujahr
Opel	2011
VW	2015
Audi	2015

In einer `SELECT` Klausel bedeutet das Symbol $*$ anstelle der Liste mit Attributnamen *alle Attribute*. Daher liefert

```
SELECT *
FROM Autos
```

die Tabelle

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

`WHERE` Klauseln

Die `WHERE` Klausel dient dazu, ein Selektionsprädikat anzugeben. Dabei verwenden wir anstelle der logischen Symbole $\neg$, $\vee$ und $\wedge$ die Schlüsselwörter `NOT`, `OR` und `AND`. Die Operanden dieser logischen Junktoren können Ausdrücke sein, in denen die Vergleichsoperatoren $<$, $<=$, $>$, $>=$, $=$ und $<>$ auftreten. Mit diesen Vergleichsoperatoren können in SQL Strings, arithmetische Ausdrücke und weitere spezielle Typen verglichen werden.

Wir betrachten wiederum die Relation `Autos` aus Beispiel 2.5. Mit der Abfrage

```
SELECT *
FROM Autos
WHERE Marke = 'OPEL' AND Baujahr > 2010
```

erhalten wir alle Autos der Marke `OPEL`, welche später als 2010 gebaut worden sind.

In SQL gibt es auch das Schlüsselwort `BETWEEN`, mit dem sich `WHERE` Klauseln vereinfachen lassen, in denen Werte kleiner-gleich und grösser-gleich als gegebene Werte spezifiziert werden. Mit

```
SELECT *
FROM Autos
WHERE Marke = 'OPEL' AND
  Baujahr BETWEEN 2000 AND 2010
```

erhalten wir also diejenigen Opel, welche zwischen 2000 und 2010 gebaut worden sind. Das Schlüsselwort `BETWEEN` schliesst die Grenzen mit ein. Die Abfrage ist also äquivalent zu

```
SELECT *
FROM Autos
WHERE Marke = 'OPEL' AND
  Baujahr >= 2000 AND Baujahr <= 2010
```

Für die nächsten Beispiele verwenden wir folgende Tabelle:

Personen

Id	Name
1	Tom
2	Eva
3	-
4	Tim
5	
6	Thom

In dieser Tabelle hat das `Name` Attribut des Tupels mit `Id` 3 den Wert `Null`. Der Wert des `Name` Attributs des Tupels mit `Id` 5 ist der leere String. Für Stringvergleiche können wir mit dem Operator `LIKE` auch Zeichenketten mit einem Muster vergleichen. In einem solchen Muster steht der Platzhalter _ für ein beliebiges Zeichen. Der Platzhalter % steht für eine beliebige (ev. leere) Zeichenkette. Somit liefert die Abfrage

```
SELECT Id
FROM Personen
WHERE Name LIKE 'T%'
```

die `Id`s aller Personen, deren Name mit `T` beginnt: 1, 4 und 6.

Die Abfrage

```
SELECT Id
FROM Personen
WHERE Name LIKE 'T_m'
```

liefert die `Id`s aller Personen, deren Namen mit `T` beginnt, dann genau ein Zeichen enthält und dann mit `m` endet: 1 und 4.

Die Abfrage

```
SELECT Id
FROM Personen
WHERE Name LIKE '%'
```

liefert die `Id`s aller Personen, die einen beliebigen (ev. leeren) Namen haben: 1, 2, 4, 5 und 6. Das Tupel 3 fehlt im Resultat, da der Vergleich

```
Null LIKE '%'
```

den Wert `unknown` liefert. Somit ergibt

```
SELECT Id
FROM Personen
WHERE Name NOT LIKE '%'
```

eine leere Tabelle (vergleiche die Definition der logischen Negation auf Seite 49).

Anmerkung 5.2. Bei Vergleichen mit dem Schlüsselwort `LIKE` gilt es zu beachten, dass diese im Allgemeinen nicht effizient durchgeführt werden können, da das Patternmatching Tupel für Tupel durchgeführt werden muss. Optimierungen sind dabei nur in Spezialfällen möglich.

In einer `WHERE` Klausel kann auch geprüft werden, ob ein Attribut den Wert `Null` hat. Dazu wird der Operator `IS NULL` eingesetzt. Die Abfrage

```
SELECT Id
FROM Personen
WHERE Name IS NULL
```

gibt also die Ids derjenigen Personen aus, deren Name nicht bekannt ist: 3.

Analog wird mit dem Operator `IS NOT NULL` getestet, ob ein Attribut einen Wert hat. Die Abfrage

```
SELECT Id
FROM Personen
WHERE Name IS NOT NULL
```

gibt also die Ids derjenigen Personen aus, deren Name bekannt ist (es kann auch der leere String sein): 1, 2, 4, 5, 6.

Anmerkung 5.3. Auch bei Abfragen dieser Form gilt es zu beachten, dass Tests auf `IS NOT NULL` üblicherweise nicht effizient durchgeführt werden können.

`FROM` Klauseln

Wie erwähnt, wird durch `FROM` Klauseln das kartesische Produkt der in der Klausel aufgeführten Relationen gebildet. Man kann damit leicht Tabellen miteinander verknüpfen und Tupel kombinieren, welche via Fremdschlüssel verbunden sind.

Wir betrachten nochmals die Tabellen 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 diesen Tabellen liefert die Abfrage

```
SELECT Vorname, Nachname, Marke
FROM Autos, Personen
WHERE FahrerId = PersId
```

das folgende Resultat:

Vorname	Nachname	Marke
Tom	Studer	Opel
Eva	Studer	Opel
Eva	Studer	VW
Eva	Meier	Audi

Wenn ein Attribut in mehreren Tabellen vorkommt, so müssen wir bei jeder Angabe des Attributs spezifizieren, aus welcher Tabelle das Attribut genommen werden soll. Wir betrachten nochmals die Tabellen aus Beispiel 4.16:

Autos

Marke	Jahrgang	PersId
Opel	2010	1
VW	1990	1
Audi	2014	-
Skoda	2014	2

Personen

PersId	Name
1	Studer
2	Meier

Mit

```
SELECT DISTINCT Autos.PersId, Marke, Jahrgang, Name
FROM Autos, Personen
WHERE Autos.PersId = Personen.PersId
```

wird der natürliche Verbund `Autos` ⋈ `Personen` berechnet:

Autos ⋈ Personen

PersId	Marke	Jahrgang	Name
1	Opel	2010	Studer
1	VW	1990	Studer
2	Skoda	2014	Meier

Umbenennungen

In SQL kann man sowohl Relationen als auch Attribute umbenennen. Dazu verwendet man das Schlüsselwort `AS` und `AS` Klauseln der Form

alter_Name `AS` *neuer_Name*.

Damit können wir die Spalten in einer Resultat-Tabelle umbenennen. Ausserdem können wir Spalten, die noch keinen Namen haben, einen Namen zuweisen. Solche Spalten entstehen zum Beispiel, wenn der Wert der Spalte durch einen arithmetischen Ausdruck berechnet wird.

Betrachten wir die folgende Relation `R`:

R

a	**b**
2	4
3	5

Die Abfrage

```
SELECT 1 AS Konstante, a AS Argument, b, a+b AS Summe
FROM R
```

liefert folgendes Resultat:

Konstante	**Argument**	**b**	**Summe**
1	2	4	6
1	3	5	8

Durch die `AS` Klauseln haben wir

1. der ersten Spalte, welche die Konstante 1 enthält, den Namen `Konstante` gegeben,
2. die zweite Spalte von `a` zu `Argument` umbenannt,
3. der letzten Spalte, welche das Resultat von `a + b` enthält, den Namen `Summe` gegeben.

Es gilt zu beachten, dass die Umbenennung erst am Schluss erfolgt. Das heisst, im arithmetischen Ausdruck `a + b` verwenden wir noch den alten Namen `a`. Der Name `Argument` ist da noch unbekannt.

Mit `AS` Klauseln können auch Relationen in der `FROM` Klausel umbenannt werden. Dies kann nötig sein, wenn dieselbe Relation mehrfach in einer `FROM` Klausel vorkommt. Folgendes Beispiel illustriert diesen Fall.

Beispiel 5.4. Wir betrachten folgende Tabelle:

VaterSohn

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

Wir wollen nun daraus die Grossvater-Enkel Relation berechnen. Dazu verwenden wir die Abfrage:

```
SELECT N1.Vater AS Grossvater, N2.Sohn AS Enkel
FROM VaterSohn AS N1, VaterSohn AS N2
WHERE N1.Sohn = N2.Vater
```

Wir erhalten folgendes Resultat:

Grossvater	Enkel
Tom	Nik
Bob	Ted
Bob	Rik
Bob	Rob

Vereinigung, Differenz und Durchschnitt

Mit dem Schlüsselwort `UNION` kann die Vereinigung von zwei Relationen berechnet werden. Die Abfrage

```
  SELECT 1 as Generationen,
         Vater AS Vorfahre,
         Sohn AS Nachfolger
  FROM VaterSohn
UNION
  SELECT 2,
         N1.Vater,
         N2.Sohn
  FROM VaterSohn AS N1, VaterSohn AS N2
  WHERE N1.Sohn = N2.Vater
```

liefert das Resultat:

Generationen	Vorfahre	Nachfolger
1	Bob	Tom
1	Bob	Tim
1	Tim	Rob
1	Tom	Ted
1	Tom	Rik
1	Ted	Nik
2	Tom	Nik
2	Bob	Ted
2	Bob	Rik
2	Bob	Rob

Die UNION-Operation liefert per Default keine Duplikate zurück. Betrachten wir die Tabellen:

U1
A
1
1
2

U2
A
1
3

Die Abfrage

```
  SELECT A FROM U1
UNION
  SELECT A FROM U2
```

resultiert in

A
1
2
3

Sollen die Duplikate in die Resultat-Tabelle aufgenommen werden, so muss das explizit mit dem Schlüsselwort UNION ALL angegeben werden. Die Abfrage

```
  SELECT A FROM U1
UNION ALL
  SELECT A FROM U2
```

liefert die Tabelle

A
1
1
2
1
3

Um die Mengendifferenz zweier Tabellen zu bilden, gibt es in SQL das Schlüsselwort `EXCEPT`. Mit der Abfrage

```
  SELECT A FROM U1
EXCEPT
  SELECT A FROM U2
```

erhalten wir die Tabelle

A
2

Sollen mehrfache Vorkommen von Tupeln berücksichtig werden, so können wir auch hier den Zusatz `ALL` verwenden. Die Abfrage

```
  SELECT A FROM U1
EXCEPT ALL
  SELECT A FROM U2
```

liefert die Tabelle

A
1
2

Mit dem Schlüsselwort `INTERSECT` können wir den Durchschnitt zweier Relation bilden. Die Query

```
  SELECT A FROM U1
INTERSECT
  SELECT A FROM U2
```

resultiert in

A
1

Auch für den Durchschnitt gibt es die Variante `INTERSECT ALL`, welche in diesem Beispiel aber dasselbe Resultat wie `INTERSECT` liefert.

Sortieren

Da die relationale Algebra auf (ungeordneten) Mengen operiert, gibt es dort keine Sortier-Operation. In SQL gibt es jedoch die `ORDER BY` Klausel, mit der Tabellen nach Attributwerten sortiert werden können. Wir betrachten die Tabelle:

Personen	
Id	**Name**
1	Tom
2	-
3	Eva

Die Abfrage

```
SELECT *
FROM Personen
ORDER BY Name ASC
```

ergibt das Resultat

Id	Name
3	Eva
1	Tom
2	-

Mit dem Schlüsselwort `ORDER BY · DESC` können wir auch absteigend sortieren. In diesem Fall kommen die `Null` Werte zuerst.

5.3 Subqueries

Innerhalb von `WHERE` Klauseln können vollständige Queries mit Hilfe der Operatoren

`IN`, `NOT IN`, `ANY`, `ALL`

definiert werden, die dann als *Subqueries* bezeichnet werden. Es sei θ eine Vergleichsoperation und S eine Query / Relation. Dann kann die Bedeutung dieser vier Schlüsselwörter informell umschrieben werden als

$$\begin{aligned} a \texttt{ IN } S &: a \in S \\ a \texttt{ NOT IN } S &: a \notin S \\ a\ \theta \texttt{ ANY } S &: \exists x(x \in S \wedge a\theta x) \\ a\ \theta \texttt{ ALL } S &: \forall x(x \in S \rightarrow a\theta x). \end{aligned}$$

Beispiel 5.5. Wir betrachten nochmals das Beispiel 2.5 mit den Relationen `Autos` und `Personen`. Wir finden diejenigen Personen, die kein Auto haben, mit folgender Abfrage:

```
SELECT *
FROM Personen
WHERE PersId NOT IN (
  SELECT FahrerId
  FROM Autos)
```

Eine Liste der ältesten Autos können wir mit der nächsten Abfrage erzeugen:

```
SELECT *
FROM Autos
WHERE Baujahr <= ALL (
  SELECT Baujahr
  FROM Autos)
```

Existenzaussagen können in SQL mit Hilfe des Schlüsselwortes `EXISTS` ausgedrückt werden. Wir verwenden dies im nächsten Beispiel.

Beispiel 5.6. Bestimme alle Personen, zu denen es eine zweite Person mit demselben Vornamen gibt.

```
SELECT *
FROM Personen
WHERE EXISTS (
  SELECT P2.*
  FROM Personen AS P2
  WHERE Personen.Vorname = P2.Vorname AND
        Personen.PersId <> P2.PersId)
```

In dieser Abfrage nimmt die Subquery Bezug auf Attribute der äusseren Query (`Personen.Vorname` und `Personen.PersId`). Eine solche Query nennen wir *korrelierte Subquery*. Die Korrelation hat zur Folge, dass die Subquery für jedes Tupel der äusseren Query neu berechnet werden muss. Entsprechend sind Abfragen mit korrelierten Subqueries häufig nicht besonders effizient. Im Vergleich dazu sind die Subqueries im Beispiel 5.5 *nicht korreliert*. Die Subquery kann einmal berechnet und anschliessend zur Auswertung der äusseren Query verwendet werden.

Mit dem Schlüsselwort `EXISTS` können wir Existenzaussagen formulieren. Um Allaussagen zu behandeln, nutzen wir die logische Äquivalenz

$$\forall x \varphi(x) \quad \Longleftrightarrow \quad \neg \exists x \neg \varphi(x)$$

aus und arbeiten mit dem Prädikat `NOT EXISTS`, das zur Negation von Existenzaussagen dient.

Beispiel 5.7. Mit Hilfe von `NOT EXISTS` Subqueries lässt sich die Divisionsoperation der relationalen Algebra in SQL implementieren. Im Beispiel 4.21 sind diejenigen Mechaniker gesucht, welche *alle* Automarken, die in der Garage vorkommen, reparieren können. Dazu müssen wir die Division

$$\texttt{Mechaniker} \div \texttt{Garage}$$

berechnen. Wir realisieren die Division mit folgender Abfrage:

```
SELECT DISTINCT Name
FROM Mechaniker AS m1
WHERE NOT EXISTS (
  SELECT *
  FROM Garage
  WHERE NOT EXISTS (
    SELECT *
    FROM Mechaniker AS m2
    WHERE (m1.Name = m2.Name) AND
          (m2.Marke = Garage.Marke)))
```

Diese Abfrage ist eine SQL Formulierung der Beschreibung der Division in (4.3). Allerdings funktioniert diese Abfrage nur korrekt, falls die Tabellen `Mechaniker` und `Garage` keine `Null` Werte enthalten. Falls `Null` Werte auftreten, so wird die Bedingung

```
(m1.Name = m2.Name) AND (m2.Marke = Garage.Marke)
```

zu unknown ausgewertet und die Abfrage als Ganzes liefert nicht das gewünschte Resultat.

Im Abschn. 5.5 werden wir noch eine alternative Implementierung der Division in SQL kennen lernen.

5.4 Joins

Im SQL Standard werden mehrere JOIN Schlüsselwörter definiert, um diverse Verbund-Operationen zu berechnen. Wir betrachten folgende Tabellen:

Links	
A	**B**
a1	b1
a2	b2

Rechts	
B	**C**
b1	c1
b3	c3

Wir können den natürlichen Join dieser beiden Tabellen berechnen mit:

```
SELECT A, B, C
FROM Links NATURAL JOIN Rechts
```

Das Resultat ist dann:

A	B	C
a1	b1	c1

Anstelle eines natürlichen Verbundes können wir einen INNER JOIN berechnen, der es erlaubt die Join-Attribute explizit anzugeben. INNER JOINs entsprechen gewissermassen Θ-Joins der relationalen Algebra. Die Abfrage

```
SELECT A, Links.B, C
FROM Links INNER JOIN Rechts ON Links.B = Rechts.B
```

liefert ebenfalls

A	B	C
a1	b1	c1

Da das Join-Attribut in beiden Tabellen des Verbundes gleich heisst und wir auf Gleichheit testen, können wir die USING Notation verwenden. Dabei geben wir nicht ein beliebiges Prädikat an (wie beim Θ-Join), sondern nur die Namen der Attribute über welche die beiden Tabellen verknüpft werden sollen. Damit ist auch klar, dass die beiden

Attribute denselben Wert haben und somit nicht beide im Resultat erscheinen müssen. Damit wird auch die SELECT Klausel einfacher. Obige Abfrage ist äquivalent zu

```
SELECT A, B, C
FROM Links INNER JOIN Rechts USING (B)
```

Tupel, für die kein Verbund-Partner gefunden wird, erscheinen nicht im Resultat von NATURAL JOIN oder INNER JOIN Abfragen. Falls solche Tupel in die Resultat-Relation aufgenommen werden sollen, muss ein sogenannter *äusserer Verbund* berechnet werden. Wir haben folgende Varianten. Die Abfrage

```
SELECT A, B, C
FROM Links LEFT OUTER JOIN Rechts USING (B)
```

liefert

A	B	C
a1	b1	c1
a2	b2	-

Umgekehrt liefert die Abfrage

```
SELECT A, B, C
FROM Links RIGHT OUTER JOIN Rechts USING (B)
```

das Resultat

A	B	C
a1	b1	c1
-	b3	c3

Es gibt auch *volle äussere Joins*, die alle Tupel enthalten. Mit der Abfrage

```
SELECT A, B, C
FROM Links FULL OUTER JOIN Rechts USING (B)
```

erhalten wir das Resultat

A	B	C
a1	b1	c1
a2	b2	-
-	b3	c3

5.5 Aggregation und Gruppierung

Im theoretischen Teil haben wir die Erweiterung der relationalen Algebra durch Gruppierung und Aggregatsfunktionen betrachtet. Diese Möglichkeiten sind in SQL ebenfalls verfügbar. Mit Hilfe von `GROUP BY` Klauseln können Tupel gruppiert werden. Auf diesen Gruppen können dann die Aggregatsfunktionen

`COUNT`, `SUM`, `MIN`, `MAX`, `AVG`

berechnet werden.

Wir betrachten nochmals das Beispiel 4.30 mit der Filmdatenbank. Die Frage nach der Durchschnittslänge der Filme pro Jahr haben wir beantwortet mit dem Ausdruck

$$\Gamma(\texttt{Filme}, (\texttt{Jahr}), \textbf{avg}, \texttt{Dauer}).$$

Der entsprechende SQL Ausdruck ist:

```
SELECT Jahr, AVG(Dauer)
FROM Filme
GROUP BY Jahr
```

Anmerkung 5.8. Bei Abfragen mit einer `GROUP BY` Klausel gilt es zu beachten, dass in der `SELECT` Klausel nur Attribute vorkommen dürfen, welche entweder in der `GROUP BY` Klausel vorkommen oder durch eine Aggregatsfunktion berechnet werden.

Der Grund dafür ist, dass für jedes Attribut der `SELECT` Klausel alle Tupel innerhalb einer Gruppe denselben Wert liefern müssen. Das lässt sich durch obige Bedingung am einfachsten erreichen.

In SQL gibt es einen Sonderfall für die `COUNT` Funktion. Anstelle von

`COUNT`(Attributname)

können wir auch

`COUNT`(*)

verwenden. Die Funktion `COUNT`(*) bestimmt die Anzahl Elemente jeder Gruppe. Dabei werden auch Tupel mitgezählt, die `Null` Werte enthalten. Mit der Abfrage

```
SELECT Jahr, COUNT(*)
FROM Filme
GROUP BY Jahr
```

können wir somit für jedes Jahr die Anzahl Filme bestimmen.

Um Abfragen mit `GROUP BY` Klauseln genauer zu studieren, verwenden wir nun folgende Instanz von `Filme`, welche Filme mit einem unbekannten Jahr enthält. Bei diesen Filmen hat das Attribut `Jahr` den Wert `Null`.

Filme

FId	Jahr	Dauer
1	2010	110
2	2012	90
3	2012	120
4	2010	100
5	2012	120
6	2011	95
7	2010	12
8	-	140
9	2012	16
10	-	130

Natürlich können SQL Abfragen mit Gruppierung auch `WHERE`-Klauseln enthalten. Diese werden dann vor der Gruppierung angewendet. Damit kann man erreichen, dass nur bestimmte Tupel in die Gruppen aufgenommen werden. Beispielsweise können wir verlangen, dass Kurzfilme (d. h. `Dauer` $<$ 30) nicht berücksichtigt werden sollen. Die Query

```
SELECT Jahr, AVG(Dauer)
FROM Filme
WHERE Dauer >= 30
GROUP BY Jahr
```

liefert die Antwort:

Jahr	AVG(Dauer)
2010	105
2011	95
2012	110
-	135

Bei dieser Abfrage werden also Kurzfilme nicht in die Gruppen aufgenommen und somit bei der Berechnung der Durchschnittslänge auch nicht berücksichtigt.

Ausserdem sehen wir an diesem Beispiel, dass die Filme, bei denen das Attribut `Jahr` den Wert `Null` hat, zu einer Gruppe zusammengefasst werden.

Nach der `GROUP BY` Klausel können wir noch eine `HAVING` Klausel angeben. Diese spezifiziert eine Bedingung, welche die einzelnen Gruppen erfüllen müssen, damit sie in die Resultat-Tabelle aufgenommen werden. Das Prädikat der `HAVING` Klausel dient also einer Selektion nach der Gruppenbildung.

Wir können beispielsweise verlangen, dass keine Jahre ins Resultat aufgenommen werden, in denen nur *ein* Film erschienen ist oder bei denen das Jahr unbekannt ist. Dies wird durch folgende Abfrage erreicht:

```
SELECT Jahr, AVG(Dauer)
FROM Filme
WHERE Dauer >= 30
GROUP BY Jahr
HAVING COUNT(*)>1 AND Jahr IS NOT NULL
```

Die Antwort ist dann:

Jahr	AVG(Dauer)
2010	105
2012	110

Die allgemeine Form eines SQL Query Blocks mit Aggregation und Gruppierung ist somit:

`SELECT` $A_1, A_2, \ldots, A_m, \mathtt{AGG}_1(A_{m+1}), \ldots, \mathtt{AGG}_k(A_{m+k})$

`FROM` $R_1, R_2, \ldots, R_n$

`WHERE` Θ_1

`GROUP BY` $A_{r_1}, \ldots, A_{r_s}$

`HAVING` Θ_2

`ORDER BY ...`

Wie früher repräsentiert dabei jedes A_i ein Attribut und jedes R_j eine Relation; Θ_1 und Θ_2 sind Prädikate. Ferner muss

$$\{A_1, A_2, \ldots, A_m\} \subseteq \{A_{r_1}, \ldots, A_{r_s}\}$$

gelten. Eine Abfrage dieser Art wird sequentiell wie folgt abgearbeitet:

1. Bedingung Θ_1 der `WHERE` Klausel auswerten.
2. Gruppierung gemäss `GROUP BY` auswerten.
3. Bedingung Θ_2 der `HAVING` Klausel auswerten.
4. Ergebnis in der Sortierfolge gemäss `ORDER BY` ausgeben.

Aggregatsfunktionen können auch ohne Gruppierung verwendet werden. In diesem Fall wird die ganze Resultat-Tabelle als eine einzige Gruppe betrachtet. Das heisst, die Aggregatsfunktionen werden über die ganze Tabelle berechnet. Beispielsweise finden wir die Laufzeit des kürzesten Films mit folgender Abfrage:

```
SELECT MIN(Dauer)
FROM Filme
```

Das Resultat ist dann 12.

Wenn wir auch die Film-Id des kürzesten Films wissen möchten, so geht das mit folgender Abfrage:

```
SELECT FId, Dauer
FROM Filme
WHERE Dauer = ( SELECT MIN(Dauer)
                FROM Filme )
```

Wir suchen zuerst mit einer Subquery die Dauer des kürzesten Films und suchen nachher alle Filme, welche diese kürzeste Dauer haben.

Anmerkung 5.9. Im obigen Beispiel verwenden wir in der `WHERE` Klausel ein Prädikat der Form

$$\texttt{A = (Subquery)}. \tag{5.1}$$

Diese Konstruktion ist nur zulässig, wenn die beiden folgenden Bedingungen erfüllt sind:

1. Das Schema der Subquery besteht aus genau einem Attribut.
2. Das Resultat der Subquery besteht aus höchstens einem Tupel.

Der Wert der Subquery `(S)` wird dann wie folgt bestimmt:

1. Falls das Resultat der Subquery `(S)` leer ist, so hat `(S)` den Wert `Null`.
2. Falls das Resultat der Subquery `(S)` aus einem 1-Tupel (b) besteht, so hat `(S)` den Wert b.

Die Verwendung der `COUNT` Funktion kann etwas tricky sein, da diese `Null` Werte nicht mitzählt (siehe Definition 4.28). Somit liefert die Abfrage

```
SELECT COUNT(Jahr)
FROM Filme
```

das Resultat 8.

Will man eine Aggregatsfunktion über einer Menge (statt über einer Multimenge) berechnen, d. h. jeden Wert nur einmal berücksichtigen, so kann man das Schlüsselwort `DISTINCT` hinzufügen.

Folgende Abfrage zählt, wie viele verschiedene Jahre in der Filmdatenbank vorkommen:

```
SELECT COUNT(DISTINCT Jahr)
FROM Filme
```

Das Resultat lautet 3, da `Null` Werte nicht mitgezählt werden.

Möchte man die Gruppe mit Jahr `Null` auch mitzählen, so wird die Abfrage komplizierter. Man könnte eine Gruppierung nach `Jahr` durchführen und dann die Anzahl Gruppen zählen. Dazu muss man eine Subquery in einer `FROM` Klausel formulieren. Folgendes Beispiel illustriert diesen Ansatz:

Beispiel 5.10. Mit dieser Abfrage finden wir die Anzahl der verschiedenen Jahre, wobei `Null` auch gezählt wird:

```
SELECT COUNT(*)
FROM (
  SELECT Jahr
  FROM Filme
  GROUP BY Jahr ) AS G
```

Diese Abfrage liefert nun das Resultat 4. Bei dieser Form von Subqueries ist zu beachten, dass man jeder Subquery in der `FROM` Klausel einen Namen geben muss. In unserem Beispiel ist das `G`.

Wir können Gruppierung und Aggregatsfunktionen verwenden, um eine alternative Abfrage für die relationale Division zu formulieren. Die Division

$$\texttt{Mechaniker} \div \texttt{Garage}$$

aus den Beispielen 4.21 und 5.7 können wir auch durch folgende Abfrage berechnen:

```
SELECT DISTINCT Name
FROM Mechaniker, Garage
WHERE Mechaniker.Marke = Garage.Marke
GROUP BY Name
HAVING COUNT(Mechaniker.Marke) = ( SELECT COUNT(Marke)
                                   FROM Garage )
```

Vergleichen wir diese Abfrage mit derjenigen aus Beispiel 5.7, so gibt es einen wichtigen Unterschied. Die beiden Abfragen verhalten sich unterschiedlich, falls die Relation `Garage` leer ist. Die Abfrage aus Beispiel 5.7 liefert in diesem Fall alle Mechaniker als Resultat. Hingegen liefert die Abfrage mit den `COUNT` Funktionen die leere Menge

als Resultat. Die Division durch 0 (d. h. durch die leere Relation) ist auch in SQL problematisch!

Als letztes Beispiel für Anwendungen von Aggregatsfunktionen betrachten wir Ranglisten. Wir haben schon gesehen, wie wir den kürzesten Film finden können. Nun geht es darum, die zwei kürzesten Filme zu finden. Wir verwenden folgende `Filme` Tabelle:

Filme

FId	Jahr	Dauer
6	2011	95
7	2010	12
8	-	140
9	2012	16
10	-	130
11	2011	16

Die zwei kürzesten Filme können wir mit folgender Abfrage bestimmen:

```
SELECT *
FROM Filme
WHERE 2 > ( SELECT count(*)
            FROM Filme AS T
            WHERE T.Dauer<Filme.Dauer )
```

Es werden also alle Filme angegeben, für die es weniger als zwei Filme mit kürzerer Dauer gibt. Das Resultat ist somit:

FId	Jahr	Dauer
7	2010	12
9	2012	16
11	2011	16

Obwohl wir die *zwei* kürzesten Filme gesucht haben, enthält die Resultat-Tabelle *drei* Tupel. Der Grund ist natürlich, dass sich zwei gleich lange Filme den 2. Platz teilen.

Anmerkung 5.11. In PostgreSQL gibt es die Möglichkeit einer Abfrage eine `LIMIT` Klausel hinzuzufügen. Damit lässt sich angeben, dass nur die ersten n-Tupel, welche die Abfrage liefert, tatsächlich in die Resultat-Tabelle aufgenommen werden sollen. Die zwei kürzesten Filme finden wir also mit:

```
SELECT *
FROM Filme
ORDER BY Dauer ASC
LIMIT 2
```

Wir erhalten das Resultat:

FId	Jahr	Dauer
7	2010	12
9	2012	16

Es erscheinen also nur zwei Filme im Resultat. Dass es noch einen weiteren Film mit der gleichen Länge gibt, wird ignoriert.

5.6 Rang Abfragen und Window Funktionen

Im vorherigen Abschnitt haben wir gesehen, wie wir mit Hilfe von Aggregatsfunktionen und Subqueries den kürzesten Film, beziehungsweise die zwei kürzesten Filme, bestimmen können. Seit einigen Jahren bietet SQL einen speziellen Mechanismus an, um solche Rang-Abfragen einfach zu formulieren. Dieser basiert auf `ORDER BY` Klauseln und einer Funktion `RANK()`, welche den Rang eines Tupels in der durch `ORDER BY` sortierten Tabelle berechnet.

Als Beispiel dient uns in diesem Kapitel die Datenbank eines Weinguts. Für jedes `Jahr` und jede `Sorte` speichern wir ab, welche `Menge` Wein produziert wurde und welche `Bewertung` dieser erhalten hatte. Wir arbeiten mit folgender Tabelle:

Wein

WId	Sorte	Jahr	Menge	Bewertung
1	weiss	2014	20000	-
2	rot	2014	7000	-
3	weiss	2013	18000	7
4	rot	2013	9000	6
5	weiss	2012	18000	8
6	rot	2012	5000	10
7	weiss	2011	14000	5
8	rot	2011	8000	6
9	weiss	2010	19000	7
10	rot	2010	6000	5

Wir wollen nun eine Rangliste der Weine gemäss ihrer Bewertung erstellen. Dabei soll die höchste Bewertung den ersten Rang belegen. Dazu können wir folgende Abfrage verwenden:

```
SELECT
  WId,
  Bewertung,
  RANK() OVER (ORDER BY Bewertung DESC NULLS LAST)
    AS Rang
FROM Wein
```

Diese liefert das Resultat:

WId	Bewertung	Rang
6	10	1
5	8	2
9	7	3
3	7	3
8	6	5
4	6	5
10	5	7
7	5	7
2	-	9
1	-	9

Das Schlüsselwort `DESC` gibt an, dass die höchste Bewertung zuerst kommen soll. Der Zusatz `NULLS LAST` sagt, dass `Null` Werte am Schluss der Sortierung erscheinen sollen. Für jedes Tupel berechnet dann die `RANK()` Funktion den Rang des Tupels in dieser Sortierung.

Eine grundlegende Frage bei Ranglisten ist, wie mit der Situation umgegangen werden soll, wenn mehrere Tupel dieselbe Bewertung haben. Die `RANK()` Funktion weist diesen Tupeln denselben Rang zu und lässt anschliessend eine entsprechende Lücke. In unserem Beispiel haben die Weine 9 und 3 beide den Rang 3. Da dieser doppelt belegt ist, wird der Rang 4 übersprungen und die Rangliste wird mit Rang 5 weitergeführt.

Es gibt auch eine Funktion `DENSE_RANK()`, welche keine Lücken in der Rangliste erzeugt (aber ebenfalls Ränge doppelt vergibt). Mit

```
SELECT
  WId,
  Bewertung,
    DENSE_RANK() OVER (ORDER BY Bewertung
                       DESC NULLS LAST)
    AS Rang
FROM Wein
```

erhalten wir das Resultat:

WId	Bewertung	Rang
6	10	1
5	8	2
9	7	3
3	7	3
8	6	4
4	6	4
10	5	5
7	5	5
2	-	6
1	-	6

Es ist auch möglich, den Rang innerhalb einer Sorte zu bestimmen. Dazu gibt es in SQL das Schlüsselwort `PARTITION BY`. Wir betrachten folgende Abfrage:

```
SELECT
  WId,
  Sorte,
  Bewertung,
  RANK() OVER (PARTITION BY Sorte
               ORDER BY Bewertung DESC NULLS LAST)
    AS Rang
FROM Wein
```

Diese liefert das Resultat:

WId	Sorte	Bewertung	Rang
6	rot	10	1
8	rot	6	2
4	rot	6	2
10	rot	5	4
2	rot	-	5
5	weiss	8	1
9	weiss	7	2
3	weiss	7	2
7	weiss	5	4
1	weiss	-	5

In PostgreSQL gibt es noch eine Reihe weiterer Funktionen, welche anstelle von `RANK()` und `DENSE_RANK()` verwendet werden können. Beispielsweise kann man mit

der Funktion `NTILE(3)` berechnen lassen, ob ein Tupel zum ersten Drittel, zweiten Drittel oder letzten Drittel einer Rangliste gehört. Diese Funktionen sind unter dem Titel *Window Funktionen* in der PostgreSQL Dokumentation beschrieben.

Der Name *Window Funktion* kommt von folgender Vorstellung: Mit

```
(PARTITION BY Sorte ORDER BY Bewertung DESC NULLS LAST)
```

wird für jedes Tupel *t* der Tabelle `Wein` ein sogenanntes Fenster (Window) bestimmt, durch das man einen Teil der Daten sieht. Das Resultat der Window Funktion wird dann aus den Daten dieses Fensters und dem Tupel *t* berechnet. In unserem Beispiel wird der Rang von *t* innerhalb des Fensters bestimmt. Dabei heisst `PARTITION BY Sorte`, dass das Fenster aus allen Weinen besteht, welche dieselbe `Sorte` wie das Tupel *t* haben. Somit wird der Rang eines Rotweins nur unter den Rotweinen bestimmt. Falls die `PARTITION BY` Klausel fehlt, enthält das Fenster *alle* Tupel (siehe das erste Beispiel in diesem Abschnitt).

Dieser Fenster-Mechanismus ist sehr mächtig. Es können nämlich nicht nur Rang-Funktionen auf einem Fester berechnet werden, sondern beliebige Aggregatsfunktionen. Ausserdem können Fenster so definiert werden, dass ein Tupel zu mehreren Fenstern gehören kann. Damit können wir beispielsweise die kumulierte Menge Wein jeder Sorte von Anfang der Datenerfassung bis zu einem bestimmten Jahr bestimmen. Dazu betrachten wir folgende Abfrage:

```
SELECT
  Jahr,
  Sorte,
  SUM(Menge) OVER (PARTITION BY Sorte
                   ORDER BY Jahr ASC
                   ROWS UNBOUNDED PRECEDING)
    AS Summe
FROM Wein
```

Als Resultat erhalten wir:

Jahr	Sorte	Summe
2010	rot	6000
2011	rot	14000
2012	rot	19000
2013	rot	28000
2014	rot	35000
2010	weiss	19000
2011	weiss	33000
2012	weiss	51000
2013	weiss	69000
2014	weiss	89000

Die Bedingung `ROWS UNBOUNDED PRECEDING` heisst, dass zum Fenster eines Tupels *t* alle Tupel gehören, welche *t* in der partitionierten und sortierten Tabelle vorangehen (inklusive *t* selbst).

Im nächsten Beispiel wollen wir die Qualitätsentwicklung der Weine bestimmen. Dazu betrachten wir nicht die einzelnen Bewertungen, sondern den Durchschnitt der Bewertungen der jeweils letzten drei Jahre. So kann die Wirkung von Ausreissern gedämpft werden. Dazu verwenden wir folgende Abfrage:

```
SELECT
  Jahr,
  Sorte,
  AVG(Bewertung) OVER (PARTITION BY Sorte
                       ORDER BY Jahr ASC
                       ROWS 2 PRECEDING)
    AS Schnitt
FROM Wein
```

Diese liefert das folgende Resultat:

Jahr	Sorte	Schnitt
2010	rot	5
2011	rot	5.5
2012	rot	7
2013	rot	7.333
2014	rot	8
2010	weiss	7
2011	weiss	6
2012	weiss	6.666
2013	weiss	6.666
2014	weiss	7.5

Die Bedingung `ROWS 2 PRECEDING` heisst, dass zum Fenster eines Tupels *t* das Tupel *t* selbst und die zwei *t* direkt vorangehenden Tupel gehören. PostgreSQL bietet noch weitere Möglichkeiten, um Fenster zu definieren. Diese sind in der Dokumentation gut beschrieben. Jedoch unterstützt PostgreSQL nicht alle Definitionsmöglichkeiten für Fenster, welche der SQL Standard vorgibt.

Wir betrachten nun noch den Fall, dass wir eine Rangliste der Jahrgänge erstellen möchten. Das heisst, wir berechnen die durchschnittliche Bewertung jedes Jahrgangs und bestimmen dann die Rangliste dieser Durchschnitte. Dazu verwenden wir folgende Abfrage:

```
SELECT
 Jahr,
 AVG(Bewertung),
 RANK() OVER (ORDER BY AVG(Bewertung) DESC NULLS LAST)
    AS Rang
FROM Wein
GROUP BY Jahr
```

Wir erhalten damit:

Jahr	Avg	Rang
2012	9	1
2013	6.5	2
2010	6	3
2011	5.5	4
2014	-	5

Es ist also möglich Gruppierung und Window Funktionen zu kombinieren. In diesem Fall wird zuerst die Gruppierung ausgeführt und die Aggregatsfunktionen auf den Gruppen berechnet. Diese können dann zur Definition der Fenster und zur Berechnung der Aggregatsfunktionen auf den Fenstern verwendet werden.

Zum Schluss kombinieren wir die beiden letzten Beispiele. Das heisst, wir betrachten nicht die durchschnittliche Bewertung eines Jahrgangs, sondern bilden den Schnitt über drei Jahre dieser durchschnittlichen Bewertungen. Folgende Abfrage bestimmt diese Werte:

```
SELECT
 Jahr,
 AVG(AVG(Bewertung)) OVER (ORDER BY Jahr ASC
                           ROWS 2 PRECEDING)
    AS Schnitt
FROM Wein
GROUP BY Jahr
```

Dabei berechnet `AVG(Bewertung)` die durchschnittliche Bewertung pro Jahr und `AVG(AVG(Bewertung))` bildet dann den Durchschnitt der im Fenster enthaltenen durchschnittlichen Bewertungen. Damit erhalten wir das Resultat:

Jahr	Schnitt
2010	6
2011	5.75
2012	6.833
2013	7
2014	7.75

5.7 WITH Klauseln und Rekursion

Im Beispiel 5.10 haben wir eine Subquery in einer FROM Klausel angegeben. Man kann nun solche Subqueries auslagern, indem man sie in einer WITH Klausel definiert. Diese ausgelagerten Subqueries können dann in der Haupt-Abfrage (mehrfach) referenziert werden. Diese Kapselung von Subqueries erhöht zum einen die Lesbarkeit der Abfrage, zum anderen können auch Code-Duplikate vermieden werden.

Die Abfrage aus Beispiel 5.10 wird mit einer WITH Klausel wie folgt definiert:

```
WITH G AS (
  SELECT Jahr
  FROM FILME
  GROUP BY Jahr )
SELECT COUNT(*)
FROM G
```

Mit dem Schlüsselwort WITH RECURSIVE ist es möglich, auch rekursive Subqueries zu definieren. Wir betrachten wiederum die VaterSohn-Relation aus Beispiel 5.4. Nun wollen wir nicht die Grossvater-Enkel Relation berechnen sondern allgemein eine Vorfahre–Nachfolger Relation über beliebig viele Generationen. Da die Anzahl Generationen beliebig sein soll, können wir das nur mit Hilfe einer *rekursiven* Berechnung erledigen. Verschiedene Datenbanksysteme haben verschiedene Syntaxregeln für rekursive Abfragen. In PostgreSQL können wir mit Hilfe einer WITH RECURSIVE Klausel folgende Query formulieren:

```
WITH RECURSIVE t(v,n) AS (
    SELECT Vater, Sohn
    FROM VaterSohn
  UNION ALL
    SELECT t.v, VaterSohn.Sohn
    FROM t INNER JOIN VaterSohn
      ON t.n = VaterSohn.Vater
  )
SELECT v AS Vorfahre, n AS Nachfolger
FROM t
```

Wir erhalten das Resultat:

Vorfahre	Nachfolger
Bob	Tom
Bob	Tim
Tim	Rob
Tom	Ted
Tom	Rik
Ted	Nik
Bob	Rik
Bob	Ted
Bob	Rob
Tom	Nik
Bob	Nik

Die Subquery, welche mit der `WITH RECURSIVE` Klausel spezifiziert wurde, ist rekursiv, denn in der Definition der Relation `t(v,n)` wird auf `t.v` und `t.n` Bezug genommen.

Weiterführende Literatur[1]

1. ANSI: Database Language SQL (2011). Dokument X3.135-2011
2. Celko, J.: Divided we stand: The SQL of relational division (2009). https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division/. Zugegriffen am 11.06.2019
3. Chamberlin, D.D., Boyce, R.F.: Sequel: A structured english query language. In: Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '74, 249–264. ACM (1974). https://doi.org/10.1145/800296.811515
4. Stonebraker, M., Rowe, L.A.: The design of postgres. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, S. 340–355. ACM (1986). https://doi.org/10.1145/16894.16888
5. Unterstein, M., Matthiessen, G.: Relationale Datenbanken und SQL in Theorie und Praxis. Springer, Berlin/Heidelberg (2012)

[1] SQL basiert auf der Abfragesprache SEQUEL, welche in den frühen 70er Jahren von IBM entwickelt wurde [3]. Das American National Standards Institute (ANSI) legte dann 1986 den ersten Standard für SQL fest. Seither gab es mehrere Aktualisierungen und Erweiterungen. Die neueste Version des Standards ist SQL:2011 [1]. Eine Schritt-für-Schritt Einführung in SQL findet sich im Buch von Unterstein und Matthiessen [5]. PostgreSQL, mit dem ursprünglichen Namen Postgres, geht auf Ingres zurück, eines der ersten relationalen Datenbanksysteme. Der Name Postgres steht denn auch für Post-Ingres. Die Basis von Postgres wurde 1986 von Stonebraker und Rowe [4] erstmals beschrieben. Eine ausgezeichnete Beschreibung der verschiedenen Möglichkeiten die Division in SQL zu implementieren findet sich in [2].