Bäume in SQL

Viele Gegebenheiten haben eine *hierarchische* Struktur. Man denke etwa an den organisatorischen Aufbau von Institutionen, an Klassifikationen im Bereich der Flora und Fauna, an Produktkataloge oder an die politische Gliederung eines Landes in Gebietskörperschaften.

In der Informatik werden solche Strukturen als *gewurzelte Bäume* modelliert. Ein gewurzelter Baum besteht aus einer Menge von *Knoten*, die durch *Kanten* verbunden sein können. Einer der Knoten ist als *Wurzel* ausgezeichnet. Ferner muss gelten: (1) der Baum ist zusammenhängend, d.h. es gibt zwischen zwei Knoten einen sie verbindenden Kantenweg von lauter verschiedenen Kanten (auch *Pfad* genannt) und (2) es gibt genau einen Pfad zwischen zwei Knoten, d.h. es gibt keinen Zyklus. Bei den oben genannten Beispielen von Hierarchien ist die oberste Instanz der Hierarchie die Wurzel. Ausgehend von der Wurzel kann man jeden anderen Knoten durch exakt einen Pfad erreichen.

Gewurzelte Bäume, im Folgenden immer kurz Bäume, kann man auch rekursiv so definieren[1]: Ein Baum besteht aus einer endlichen Menge von Knoten mit folgenden Eigenschaften

a) Einer der Knoten ist als Wurzel des Baums ausgezeichnet.

b) Die restliche Menge an Knoten zerfällt in paarweise disjunkte Teilmengen von Knoten, die selbst wieder einen Baum bilden, die *Teilbäume* der Wurzel.

c) Die Teilbäume der Wurzel sind durch eine Kante mit der Wurzel verbunden.

Da Bäume als Datenstruktur in der Informatik häufig verwendet werden, müssen sie auch *persistiert* werden. Dafür werden in der Regel Datenbanken verwendet. Es gibt Datenmodelle (und sie implementierende Datenbanken), die speziell Graphen oder hierarchisch aufgebaute Dokumente unterstützen. Die gerade in Geschäftsanwendungen jedoch am häufigsten eingesetzten Datenbanken basieren auf dem relationalen Datenmodell und verwenden die Datenbanksprache *Structured Query Language (SQL)*.

Wir wollen hier untersuchen, wie man Bäume in SQL-Datenbanken speichern kann. Als laufendes Beispiel dient ein Ausschnitt aus der politischen Gliederung der Bundesrepublik und auch Frankreichs, die in Abb. 1 und Abb. 2 dargestellt sind. (Die Abkürzung „RB“ steht für „Regierungsbezirk“.)

Im folgenden Abschnitt betrachten wir zunächst die aus der Definition eines Baums folgende *natürliche* Modellierung im relationalen Datenmodell.

[1] Mehr zur rekursiven Definition eines Baums in [2, S. 308ff]

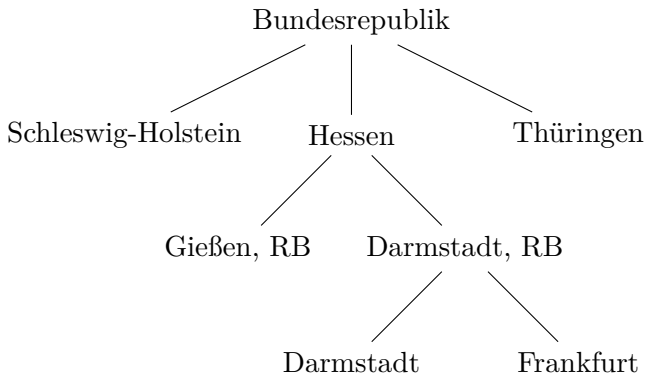

Abbildung 1: Ausschnitt der politischen Gliederung der BRD

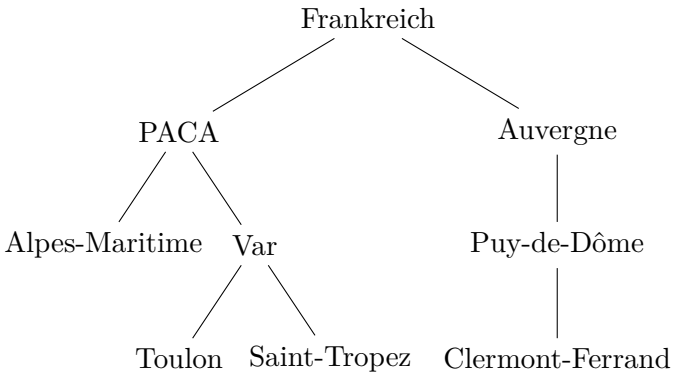

Abbildung 2: Ausschnitt der politischen Gliederung Frankreichs

Bäume

Jeder Teilbaum eines Baums in der obigen rekursiven Definition des Baums ist durch eine Kante mit seinem übergeordneten Knoten verbunden.[2] Wenn wir nun jeden Knoten des Baums mit einer `id` identifizieren, dann können wir einem Knoten auch seinen übergeordneten Knoten zuordnen. Dazu versehen wir jeden Knoten mit einem Attribut `pid`, was für *parentId* steht. Es ist in der Informatik üblich, die Beziehungen in Bäumen analog zu Familienbeziehungen zu bezeichnen: Man spricht vom Elternteil (oft auch Vater) eines Knotens, von Geschwistern, von Vorfahren und von Nachkommen [2, S. 311].

In folgendem Listing[3] wird eine Tabelle definiert, die einen Baum in der

[2] Dieses Konzept wird in der Literatur oft als *Adjacency List Model* bezeichnet, siehe etwa [1, Kapitel 2]. Dies trifft es nicht so ganz, denn Adjazenzlisten enthalten normalerweise für ungerichtete Graphen *alle* Nachbarn. Wir speichern aber keine Liste, sondern genau einen Nachbarn, nämlich den Elternknoten.

[3] Alle Beispiele mit SQL wurden mit PostgreSQL gemacht.

„natürlichen“ Modellierung in SQL enthalten kann. Als Merkmale der Knoten werden in unseren einfachen Beispiele nur die `id` und das Attribut `name` verwendet. In Anwendungen haben die Knoten typischerweise weitere Attribute. Man kann sich auch ein Datenmodell vorstellen, in dem die zu den Knoten gehörenden zusätzlichen Angaben durch eine Referenz auf Datensätze einer anderen Tabelle zugeordnet werden.

```
create table tree (
  id int primary key,
  pid int references tree(id),
  name varchar(128)
);
```

Um nun in einer so definierten Tabelle einen Baum zu speichern, wird man beginnend mit der Wurzel Datensätze mit einer eindeutigen `id` speichern, die jeweils im Attribut `pid` die `id` des Elternknotens enthalten. Bei der Wurzel gibt es keinen Elternknoten, deshalb bleibt das Attribut `pid` der Wurzel undefiniert, in SQL also `null`.

Für unseren Ausschnitt der politischen Gliederung der BRD sieht die Tabelle aus wie in Tabelle 1.

Tabelle 1: Inhalt der Tabelle mit dem Ausschnitt der politischen Gliederung der BRD

id	pid	name
1	<null>	Bundesrepublik
2	1	Schleswig-Holstein
3	1	Hessen
4	1	Thüringen
5	3	Gießen, RB
6	3	Darmstadt, RB
7	6	Darmstadt
8	6	Frankfurt

Folgende Aussagen kann man diesem Modell unmittelbar entnehmen:

1. Es ist sehr einfach zu einem Eintrag den Elternknoten oder die Kinder zu finden.
2. Will man zu einem Eintrag alle Vorfahren oder alle Nachkommen finden, dann muss man den via `pid` verbundenen Einträgen *rekursiv* folgen. Dies ist in SQL mit sogenannten *recursive common table expressions*[4] möglich.

[4] Recursive CTEs wurden in SQL:1999 standardisiert.
In manchen Datenbankmanagementsystemen hat es sehr lange gedauert, bis das Konzept implementiert wurde. In MySQL etwa gibt es CTEs erst seit Version 8, siehe MySQL 8.0 Reference Manual 13.2.20 WITH (Common Table Expressions).
Man spricht wegen der syntaktischen Umsetzung des Konzepts der CTEs in SQL auch

3. Manipulationen des Baums, wie das Hinzufügen von Knoten oder das Verschieben von Teilbäumen, sind einfach umzusetzen.
4. Will man jedoch Teilbäume löschen, muss man alle Nachkommen finden, siehe 2.

Wir werden weiter unten genauer analysieren, wie man diese Aufgaben umsetzen kann.

Doch zunächst wollen wir einen zweiten Baum in unserer Tabelle speichern, nämlich die Daten aus Frankreich. Dazu legt man einfach eine neue Wurzel an und schon beherbergt unser Tabelle nicht nur einen Baum, sondern einen ganzen Wald (mit 2 Bäumen in unserem Beispiel), siehe Tabelle 2.

Tabelle 2: Inhalt der Tabelle ergänzt um Angaben zu Frankreich

id	pid	name
1	<null>	Bundesrepublik
2	1	Schleswig-Holstein
3	1	Hessen
4	1	Thüringen
5	3	Gießen, RB
6	3	Darmstadt, RB
7	6	Darmstadt
8	6	Frankfurt
10	<null>	Frankreich
11	10	PACA
12	10	Auvergne
13	11	Var
14	11	Alpes-Maritime
15	13	Toulon
16	13	Saint-Tropez
17	12	Puy-de-Dôme
18	17	Clermont-Ferrand

In der Definition des Baums spielt die Reihenfolge der Kindknoten eines Knotens keine Rolle. Nach unserer bisherigen Definition wäre ein Baum, bei dem Hessen links (und nicht rechts) von Schleswig-Holstein dargestellt wird identisch mit dem Baum in Abb. 1. Eine Sortierung der Knoten auf einer Ebene könnte sich natürlich aus den Angaben ergeben, die wir mit den Knoten speichern. Im Fall der Gebietskörperschaften der BRD könnte man mit den Knoten den Allgemeinen Regionalschlüssel (siehe Destatis ARS) speichern und ihn zur Sortierung der Knoten verwenden. Es kann aber sein,

von *WITH Queries*, siehe zum Beispiel PostgreSQL Dokumentation 7.8. WITH Queries (Common Table Expressions). Beispiele für den Einsatz solcher Abfragen gibt es auch in meinem Vorlesungsskript über Rekursion in SQL.

dass es kein solches Kriterium gibt und man beim Speichern des Baums selbst die Reihenfolge der Kindknoten eines Knotens festlegen möchte.

Diese Fragestellung führt uns zu *geordneten Bäumen*.

Geordnete Bäume

In einem *geordneten Baum* spielt die Anordnung der Teilbäume unter einem Knoten eine Rolle. Man spricht dann auch von *planaren Bäumen*, weil es relevant wird, wie der Baum in der Ebene dargestellt wird (Ist Hessen links oder rechts von Schleswig-Holstein in einer planaren Darstellung wie in Abb. 1 gezeichnet?) [2, S. 308f].

Eine Möglichkeit im Datenmodell einen geordneten Baum darzustellen ist das Konzept der sogenannten *nested sets* (verschachtelte Mengen), siehe [2, S. 312] und [1, Kapitel 4].

Zum Verständnis dieses Modells wechseln wir die Repräsentation des Baums und stellen uns die Knoten als XML-Elemente vor. Die Angaben zu einem Knoten werden zu XML-Attributen und jeder Knoten hat ein Anfangstag `<node>` und ein korrespondierendes Endetag `</node>`, auch wenn es sich um ein Blatt des Baums handelt, das keine Kindknoten hat. Hat ein Knoten Kindknoten, so werden diese als Unterelemente des XML-Elements des Knotens dargestellt. Auf diese Weise bekommen wir eine Repräsentation des Baums durch die Verschachtelung der XML-Elemente.

Außerdem schreiben wir uns Zeilennummern zu unserer XML-Darstellung dazu. Auf diese Weise erhalten wir die folgende Darstellung:

```
 1  <node id='1' name='Bundesrepublik'>
 2    <node id='2' name='Schleswig-Holstein'>
 3    </node>
 4    <node id='3' name='Hessen'>
 5      <node id='5' name='Gießen, RB'>
 6      </node>
 7      <node id='6' name='Darmstadt, RB'>
 8        <node id ='7' name='Darmstadt'>
 9        </node>
10        <node id ='8' name='Frankfurt'>
11        </node>
12      </node>
13    </node>
14    <node id='4' name='Thüringen'>
15    </node>
16  </node>
```

In deser Darstellung können wir nun die *Verschachtelung* direkt ablesen: Die Bundesrepublik auf der äußersten Ebene umfasst die Länder Schleswig-Holstein, Hessen und Thüringen mit all deren Unterelementen, das sind die Zeile `1` bis `16`. Das Land Hessen umfasst die Zeilen `4` bis `13` und hat als Unterelemente die beiden Regierungsbezirke Gießen und Darmstadt in dieser

Reihenfolge und der Regierungsbezirk Darmstadt hat als Kindelemente die beiden Städte Darmstadt und Frankfurt.

Diese Darstellung enthält also nicht nur die Baumstruktur, sondern auch eine Reihenfolge der Kinder jeden Knotens entsprechend der Reihenfolge in der XML-Anordnung.

Wie kann man nun diese Darstellung in einer Tabelle speichern? Hierzu nehmen wir Zeilennummern in der XML-Darstellung. Die Wurzel hat ihr Anfangstag auf Zeile `1` und ihr Endetag auf Zeile `16`. Das bedeutet, dass alle Zeilen dazwischen zu Nachkommen der Wurzel gehören. Ein weiteres Beispiel: Alle Nachkommen des Regierungsbezirks Darmstadt befinden sich in den Zeilen `7` bis `12`.

Speichert man die Zeilennummern, dann kann man daraus die XML-Darstellung mit der korrekten Reihenfolge aller Kindknoten konstruieren. In der XML-Darstellung ist die Anfangszeile eines Knotens *oberhalb* seiner Endzeile. In der Umsetzung des Konzepts in einem Tabellenschema hat sich jedoch die Sprechweise eingebürgert von der *linken* und der *rechten* Position eines Knotens zu sprechen. Wahrscheinlich resultiert diese Sprechweise aus der Darstellung als Intervalle, wie in Abb. 3.

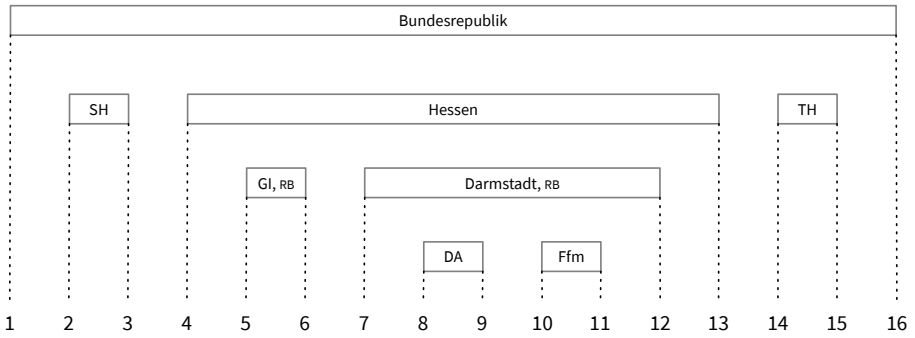

Abbildung 3: Baum in Intervall-Darstellung

Setzt man dieses Modell in das Schema einer Tabelle in SQL um, erhält man:

```
create table tree(
  id int primary key,
  lft int not null,
  rgt int not null,
  name varchar(128)
);
```

Der Inhalt dieser Tabelle wird in Tabelle 3 und Tabelle 4 dargestellt.

Die Tabelle sortiert nach dem Attribut lft entspricht gerade dem Durchlaufen des Baums in der Reihenfolge der Tiefensuche.

Dieses Konzept, einen Baum zu speichern, hat folgende Eigenschaften:

Tabelle 3: Tabelle für geordneten Baum (sortiert nach id)

id	lft	rgt	name
1	1	16	Bundesrepublik
2	2	3	Schleswig-Holstein
3	4	13	Hessen
4	14	15	Thüringen
5	5	6	Gießen, RB
6	7	12	Darmstadt, RB
7	8	9	Darmstadt
8	10	11	Frankfurt

Tabelle 4: Tabelle für geordneten Baum (sortiert nach lft)

id	lft	rgt	name
1	1	16	Bundesrepublik
2	2	3	Schleswig-Holstein
3	4	13	Hessen
5	5	6	Gießen, RB
6	7	12	Darmstadt, RB
7	8	9	Darmstadt
8	10	11	Frankfurt
4	14	15	Thüringen

1. Es ist sehr einfach, alle Vorfahren oder alle Nachkommen eines Knotens zu finden. Die Nachkommen müssen ihre linke Position zwischen der eigenen linken und rechten Position haben. Die Vorfahren sind alle diejenigen Knoten, bei denen die eigene linke Position in deren Intervall liegt.
2. Die Wurzel hat `1` als linke Position. Blätter sind charakterisiert dadurch, dass die Differenz der rechten und der linken Position gerade `1` ist.
3. Direkte Kinder oder den Elternknoten zu finden, ist in diesem Modell komplizierter.
4. Änderungen am Baum, wie das Einfügen neuer Knoten oder das Verschieben von Teilbäumen erfordert Änderungen an vielen Einträgen der Tabelle, da sich in der Regel viele Positionsangaben ändern.

Am Beispiel des Baums im ersten Abschnitt haben wir nicht nur einen Baum in der Tabelle gespeichert, sondern auch noch einen zweiten, im Beispiel einen Ausschnitt der politischen Gliederung Frankreichs. Das kann man in diesem Modell des geordneten Baums auch, zum Beispiel indem man für die Wurzel als linke Position eine nimmt, die größer ist als die höchste rech-

te Position, dis bisher vorkommt. Dann verliert man jedoch die Eigenschaft, dass eine Wurzel als linke Position die `1` hat. In der Praxis wird gerne so verfahren, dass man für jeden Baum eine eigene Kennung vergibt, oft `scope` genannt. Dies wollen wir auch so machen.

Wie bereits angedeutet, ist es relativ kompliziert im Konzept der *nested set*s die unmittelbaren Nachbarn eines Knotens zu finden. In manchen Umsetzungen wird deshalb auch die Länge des Wegs von einem Knotens zur Wurzel als Attribut in den Knoten gespeichert.

Da wir gerne eine Kombination des „natürlichen“ Modells eines Baums mit der Ordnungsinformation der *nested set*s verwenden wollen, definieren wir das folgende *hybride* Tabellenschema — wohl wissend, dass wir damit eine gewisse Redundanz in Kauf nehmen.

```
create table tree (
  id int primary key,
  pid int references tree(id),
  scope int not null,
  lft int not null,
  rgt int not null,
  name varchar(128)
);
```

Tabelle 5 zeigt den Inhalt der Tabelle für unser Beispiel mit den Daten von Deutschland und Frankreich.

Tabelle 5: Tabelle mit Daten von Deutschland und Frankreich

id	pid	scope	lft	rgt	name
1	<null>	1	1	16	Bundesrepublik
2	1	1	2	3	Schleswig-Holstein
3	1	1	4	13	Hessen
5	3	1	5	6	Gießen, RB
6	3	1	7	12	Darmstadt, RB
7	6	1	8	9	Darmstadt
8	6	1	10	11	Frankfurt
4	1	1	14	15	Thüringen
10	<null>	2	1	18	Frankreich
11	10	2	2	11	PACA
14	11	2	3	4	Alpes-Maritime
13	11	2	5	10	Var
15	13	2	6	7	Toulon
16	13	2	8	9	Saint-Tropez
12	10	2	12	17	Auvergne
17	12	2	13	16	Puy-de-Dôme
18	17	2	14	15	Clermont-Ferrand

In den folgenden beiden Abschnitten werden wir untersuchen, wie man in

geordneten Bäumen suchen kann und wie man sie verändert. Dazu verwenden wir das eben vorgestellte Datenmodell und als Beispiele unsere Daten zur politischen Gliederung der BRD bzw. Frankreichs.

Suche in geordneten Bäume

Wurzel und Blätter

Wurzel finden

Die Wurzeln der Bäume in der Tabelle sind charakterisiert durch:

1. Die Id `pid` des Elternknotens ist `<null>`, oder alternativ
2. die linke Position `lft` im *nested set* ist `1`

Also gibt es die folgenden beiden Möglichkeiten *alle* Wurzeln zu finden:

```
-- Alle Wurzeln finden

select * from tree where pid is null;
select * from tree where lft = 1;
```

Da in unserem Modell jeder Baum in der Tabelle durch das Attribut `scope` identifiziert wird, finden wir einen speziellen Baum durch die Vorgaben des Werts von `scope`. Wieder gibt es zwei Möglichkeiten.

Wir schreiben in den folgenden Listings in SQL-Anweisungen solche Vorgaben in der Form `?<name>` mit dem jeweiligen Namen des Parameters, um sie von den Bezeichnungen der Attribute der Tabelle zu unterscheiden.

```
-- Eine bestimmte Wurzel finden, vorgegeben scope

select * from tree where scope = ?scope and pid is null;
select * from tree where scope = ?scope and lft = 1;
```

Blätter finden

Blätter sind charakterisiert durch:

1. Ihre Id kommt in keinem anderen Knoten als `pid` vor, oder alternativ
2. der Abstand ihrer linken und rechten Position ist 1.

Also haben wir wieder zwei Möglichkeiten in einem durch den Parameter `scope` identifizierten Baum alle Blätter zu finden:

```
-- Alle Blätter finden, vorgegeben scope

select * from tree
  where scope = ?scope
    and id not in (select pid from tree where pid is not null);
select * from tree
  where scope = ?scope and rgt - lft = 1;
```

Unmittelbare Nachbarn

Die unmittelbaren Nachbarn eines Knotens im Baum sind der Elternknoten bzw. die Kindknoten. Wieder gibt es zwei Möglichkeiten, diese Ergebnisse zu erzielen. Die eine verwendet die Referenzierung des Elternknotens und ist damit deutlich einfacher als die zweite, die die Positionsangaben einsetzt.

Für diese Abfragen geben wir den Baum mit `scope` vor sowie die Id des Knotens, dessen unmittelbare Nachbarn wir finden wollen.

Elternknoten finden

```
-- Zu einem Knoten den Elternknoten finden, vorgegeben scope und id

select * from tree
  where id = (select pid from tree where id = ?id);
-- oder mit Join:
select tp.* from tree tc join tree tp on tp.id = tc.pid
  where tc.id = ?id;

select t2.*
  from tree t1, tree t2
  where t1.scope = ?scope and t2.scope = t1.scope
    and t2.lft < t1.lft and t2.rgt > t1.rgt
    and t1.id = ?id
  order by t2.rgt - t2.lft
  limit 1;
```

Kindknoten finden

```
-- Zu einem Knoten die Kinder finden, vorgegeben scope und id

select * from tree where pid = ?id order by lft;

select * from tree where id in (
  select child.id from tree child, tree parent
    where parent.scope = ?scope and child.scope = parent.scope
     and parent.lft < child.lft and parent.rgt > child.rgt
    group by child.id
    having max(parent.lft) = (select lft from tree
                                where scope = ?scope and id = ?id)
  order by tree.lft);
```

Die SQL-Anweisung, die mittels der linken und rechten Position der Knoten die Kinder eines Knoten findet, ist recht komplex. Um diese Anweisung zu verstehen, ist es am besten, sie sukzessive aufzubauen und in einer SQL-Shell interaktiv auszuprobieren. Die dafür interessanten Schritte sind im SQL-Skript Bäume in SQL dargestellt. In diesem Skript werden alle hier vorgestellten SQL-Anweisungen an Beispielen ausgeführt.

Vorfahren und Nachkommen

Nun möchten wir zu einem vorgegebenen Knoten *alle* Vorfahren finden sowie *alle* Nachkommen, d.h. den Teilbaum, der den gegebenen Knoten als Wurzel hat. In diesem Fall sind die SQL-Anweisungen einfacher, wenn wir die Positionen verwenden, andernsfalls benötigt man SQL-Anweisungen mit `with recursive`.

Alle Vorfahren finden

Wir haben wieder die Id des aktuellen Knotens sowie den `scope`, also die Identifizierung des Baums in unserer Tabelle gegeben.

```
-- Zu einem Knoten alle Vorfahren finden
with recursive rt as (
  select * from tree where id = ?id -- aktueller Knoten
  union
  select p.* from tree p join rt on rt.pid = p.id
  -- diejenigen, deren id die pid der bereits aufgesammlten ist
)
select * from rt
  where id <> ?id; -- wenn man den aktuellen Knoten nicht im Ergebnis will

select * from tree
  where scope = ?scope
    and (select lft from tree where id = ?id) between lft and rgt
    and id <> ?id; -- wenn man den aktuellen Knoten nicht im Ergebnis will
```

Alle Nachkommen finden

Wir gehen wieder vom aktuellen Knoten mit bekannter Id aus und kennen auch den `scope` des Baums, in dem wir suchen. Ausgehend vom aktuellen Knoten werden alle Nachkommen gesucht inklusive des Knotens selbst, also der Teilbaum mit dem aktuellen Knotens als Wurzel.

Möchte man den aktuellen Knoten selbst nicht in der Ergebnismenge haben, kann man ihn wie oben durch die Erweiterung der Where-Bedingung ausschließen.

```
- Zu einem Knoten alle Nachkommen finden = den Teilbaum dieses Knotens finden

with recursive rt as (
  select * from tree where id = ?id
  union
  select t.* from tree t join rt on rt.id = t.pid
  -- diejenigen, deren pid die id der bereits aufgesammelten ist
)
select * from rt
  order by lft;
```

```
select * from tree
  where scope = ?scope
    and lft between
      (select lft from tree where scope = ?scope and id = ?id) and
      (select rgt from tree where scope = ?scope and id = ?id)
  order by lft;
```

Im folgenden Abschnitt wird untersucht, wie man Änderungen an Bäumen in SQL machen kann.

Manipulation geordneter Bäume

Da wir uns für die Implementierung von Bäumen in SQL für ein hybrides Datenmodell entschieden haben, das sowohl Referenzen auf Elternknoten als auch linke und rechte Positionen für die Verschachtelung von Knoten enthält, müssen wir auch beide Konzepte berücksichtigen, wenn es um die Manipulation von Bäumen geht.

Uns interessieren hier natürlich nur Änderungen, die die Struktur des Baums betreffen. Änderungen an den Angaben zu den Knoten wie in unserem Beispiel des Werts von Attribut `name` haben keine Auswirkung auf die Baumstruktur. Interessante Modifikationen sind:

- Einfügen einer neuen Wurzel
- Einfügen eines neuen Knotens
- Verschieben von Teilbäumen
- Löschen von Teilbäumen

Für die folgenden Listings wird folgende Notation verwendet. Mit `new`, `target` und `sub` werden Knoten bezeichnet. Die Attribute dieser Knoten sind dann `new.id`, `new.pid` usw. Und wenn die Werte dieser Attribute in SQL-Anweisungen verwendet werden, dann bekommen sie ein Fragezeichen wie etwa `?new.id`.

Wenn verlangt wird, dass ein Wert verwendet werden muss, der bisher für das entsprechende Attribut nicht vorhanden ist, dann sagen wir, dass der Wert *frisch* sein muss.

Einfügen einer neuen Wurzel

Ein neuer Knoten `new` soll in die Tabelle `tree` eingefügt werden.

```
-- neue Wurzel new einfügen
-- Voraussetzung: new.id und new.scope sind frisch
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, null, ?new.scope, 1, 2, ?new.name);
```

Ein neuer Knoten `new` soll als Elternknoten der Wurzel eines bereits vorhandenen Baums eingefügt werden

```
-- new als Wurzel oberhalb der bisherigen Wurzel einfügen
-- Voraussetzung: new.id ist frisch und scope kommt bereits vor
-- (1) Platz schaffen
update tree set lft = lft + 1 where scope = ?scope;
update tree set rgt = rgt + 1 where scope = ?scope;
-- (2) Neue Wurzel einfügen
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, null, ?scope, 1,
    (select max(rgt) from tree where scope = ?scope) + 1, ?new.name);
-- (3) Bisherige Wurzel einhängen
update tree set pid = ?new.id
  where scope = ?scope and lft = 2;
```

Einfügen eines neuen Knotens

Abbildung 4: Mögliche Positionen relativ zum Zielknoten

Für das Einfügen eines neuen Knotens muss man angeben, bei welchem bereits vorhandenen Knoten, dem Zielknoten, er eingefügt werden soll und wo dort. Für das „wo“ gibt es vier Möglichkeiten, illustriert beispielhaft in Abb. 4:

1. Der neue Knoten wird das erste Kind des Zielknotens – `FIRST_CHILD`,
2. er wird das letzte Kind des Zielknotens – `LAST_CHILD`,
3. er wird das linke Geschwister des Zielknotens – `LEFT_SIBLING`, oder
4. er wird das rechte Geschwister des Zielknotens – `RIGHT_SIBLING`.

Die Strategie besteht beim Einfügen eines neuen Knotens immer darin, dass man (1) die Lücke für den neuen Knoten schafft, d.h. die Positionen der folgenden Knoten um `+ 2` verschiebt und dann (2) den neuen Knoten einfügt.

`new` als erstes Kind von `target` einfügen

Die Lücke muss hinter der linken Position des Zielknotens geschaffen werden.

```
-- neuen Knoten als erstes Kind einfügen
-- Voraussetzung: new.id frisch, target gegeben

-- (1) Platz schaffen
update tree set lft = lft + 2
  where scope = ?target.scope and lft > ?target.lft;
update tree set rgt = rgt + 2
  where scope = ?target.scope and rgt > ?target.lft;

-- (2) neuen Knoten einfügen
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, ?target.id, ?target.scope,
    ?target.lft + 1, ?target.lft + 2, ?new.name);
```

`new` als letztes Kind von `target` einfügen

Die Lücke muss vor der rechten Position des Zielknotens geschaffen werden.

```
-- neuen Knoten als letztes Kind einfügen
-- Voraussetzung: new.id frisch, target gegeben

-- (1) Platz schaffen
update tree set lft = lft + 2
  where scope = ?target.scope and lft > ?target.rgt - 1;
update tree set rgt = rgt + 2
  where scope = ?target.scope and rgt > ?target.rgt - 1;

-- (2) neuen Knoten einfügen
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, ?target.id, ?target.scope,
    ?target.rgt, ?target.rgt + 1, ?new.name);
```

`new` als linkes Geschwister von `target` einfügen

Die Lücke muss vor der linken Position des Zielknotens geschaffen werden.

```
-- neuen Knoten als linkes Geschwister einfügen
-- Voraussetzung: new.id frisch, target gegeben,
--                target ist nicht die Wurzel

-- (1) Platz schaffen
update tree set lft = lft + 2
  where scope = ?target.scope and lft > ?target.lft - 1;
update tree set rgt = rgt + 2
  where scope = ?target.scope and rgt > ?target.lft - 1;

-- (2) neuen Knoten einfügen
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, ?target.pid, ?target.scope,
    ?target.lft, ?target.lft + 1, ?new.name);
```

new als rechtes Geschwister von target einfügen

Die Lücke muss nach der rechten Position des Zielknotens geschaffen werden.

```
-- neuen Knoten als rechtes Geschwister einfügen
-- Voraussetzung: new.id frisch, target gegeben
--                target ist nicht die Wurzel

-- (1) Platz schaffen
update tree set lft = lft + 2
  where scope = ?target.scope and lft > ?target.rgt;
update tree set rgt = rgt + 2
  where scope = ?target.scope and rgt > ?target.rgt;

-- (2) neuen Knoten einfügen
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, ?target.pid, ?target.scope,
    ?target.rgt + 1, ?target.rgt + 2, ?new.name);
```

Pseudocode

Die Implementierung des Konzepts erfordert

1. dass die Teilschritte in einer Transaktion mit dem Isolationslevel *serializable* durchgeführt werden,
2. dass die erwähnten Voraussetzungen geprüft werden und
3. wenn das Konzept im Kontext eines Systems des objekt-relationalen Mappings eingesetzt wird, muss nach der Transaktion, die die Datenbank ändert, eine Synchronisation zwischen Datenbank und Cache des objekt-relationalen Systems durchgeführt werden.

Diese Punkte werden hier nicht behandelt, wir konzentrieren uns auf den Kern des Verfahrens.

Für den ersten Schritt beim Einfügen neuer Knoten kann man eine Funktion erstellen:

```
-- Alle linken und rechten Positionen ab from um delta verschieben

function makeSpace(from, delta, scope) {
  update tree set lft = lft + ?delta
    where scope = ?scope and lft > ?from;
  update tree set rgt = rgt + ?delta
    where scope = ?scope and rgt > ?from;
}
```

Die Position `from`, an der man die Lücke machen muss, hängt davon ab wo der neue Knoten relativ zum Zielknoten eingefügt werden soll. Also

```
-- (1)
-- bestimme from: die Position, ab der die Lücke entstehen muss
case (zielpos) {
  FIRST_CHILD:   from := target.lft
  LAST_CHILD:    from := target.rgt - 1
  LEFT_SIBLING:  from := target.lft - 1
  RIGHT_SIBLING: from := target.rgt
  }
-- bilde die Lücke mit Platz für die beiden Stellen des neuen Knotens
makeSpace(from, 2, target.scope)
```

Nun kann man den neuen Knoten in die geschaffene Lücke einfügen. Dabei muss man die `pid` entsprechend der Position relativ zum Zielknoten setzen.

```
-- (2)
-- ermittele die Id des Elternknotens
case (zielpos) {
  FIRST_CHILD:   new.pid := target.id
  LAST_CHILD:    new.pid := target.id
  LEFT_SiBLING:  new.pid := target.pid
  RIGHT_SIBLING: new.pid := target.pid
  }
-- füge den neuen Knoten ein
insert into tree(id, pid, scope, lft, rgt, name)
  values(?new.id, ?new.pid, ?target.scope, ?from + 1, ?from + 2, ?new.name);
```

Verschieben von Teilbäumen

Für das Verschieben von Teilbäumen gibt man genau wie beim Einfügen neuer Knoten den Zielknoten an und die relative Position des Ziels an diesem Knoten, ganz wie in Abb. 4 mit `sub` an Stelle von `new`.

In diesem Abschnitt wollen wir das Vorgehen zunächst an einem Beispiel demonstrieren. Danach wird der Pseudocode für das Verschieben von Teilbäumen dargestellt.

Das Verschieben von Teilbäumen hat folgende Voraussetzungen:

1. Der Zielknoten darf nicht selbst Knoten im Teilbaum sein, der verschoben werden soll, denn sonst würde ja ein Zyklus entstehen.
2. Der Teilbaum und der Zielknoten müssen denselben `scope` haben. Das Verschieben ist nur innerhalb eines Baums in unserer Tabelle erlaubt. (Es wäre möglich, auf diese Voraussetzung zu verzichten, dies würde das Vorgehen aber noch etwas aufwändiger machen.)
3. Der Zielknoten darf nicht die Wurzel sein, wenn die relative Position am Zielknoten `LEFT_SIBLING` oder `RIGHT_SIBLING` ist. Die Wurzel hat keine Geschwister.

Beispiel für das Verschieben eines Teilbaums

In diesem Beispiel beginnen wir mit unserem Ausschnitt aus der politischen Gliederung der BRD wie in Tabelle 6.

Tabelle 6: Ausgangssituation für das Verschieben eines Teilbaums

id	pid	scope	lft	rgt	name
1	<null>	1	1	16	Bundesrepublik
2	1	1	2	3	Schleswig-Holstein
3	1	1	4	13	Hessen
5	3	1	5	6	Gießen, RB
6	3	1	7	12	Darmstadt, RB
7	6	1	8	9	Darmstadt
8	6	1	10	11	Frankfurt
4	1	1	14	15	Thüringen

In diesem Beispiel verschieben wir den Teilbaum von Hessen und machen ihn zum ersten Kind der Wurzel Bundesrepublik.

Wir haben also gegeben den Knoten `sub` mit der `sub.id 3` für Hessen und als Zielknoten `target` die Wurzel mit der Id `1`. Die relative Position, an die wir Hessen verschieben wollen ist `FIRST_CHILD`.

Schritt 1: `pid` des Teilbaums ändern

```
update tree set pid = 1 where id = 3;
```

In diesem konkreten Beispiel wäre dieser Schritt nicht erforderlich, weil Hessen ja bereits ein Kind der Bundesrepublik ist. Im Allgemeinen muss man aber die `pid` des Teilbaums ändern.

Schritt 2: Lücke am Ziel des Verschiebens schaffen Dazu bestimmen wir die Zielposition. In unserem Beispiel soll der Teilbaum das erste Kind des Zielknotens werden, d.h. die Lücke muss hinter `target.lft = 1` sein. Die Größe der Lücke `delta` ergibt sich zu

```
delta = sub.rgt - sub.left + 1 = 13 - 4 + 1 = 10
```

In unserem Beispiel hat der Teilbaum Hessen insgesamt 5 Knoten, also 10 Stellen für die linken und rechten Positionen dieser Knoten.

```
-- Die Lücke schaffen
update tree set lft = lft + 10 where scope = 1 and lft > 1;
update tree set rgt = rgt + 10 where scope = 1 and rgt > 1;
```

Tabelle 7 zeigt das hierdurch entstandene Zwischenergebnis.

Tabelle 7: Zwischenergebnis nach Schritt 2

id	pid	scope	lft	rgt	name
1	<null>	1	1	26	Bundesrepublik
2	1	1	12	13	Schleswig-Holstein
3	1	1	14	23	Hessen
5	3	1	15	16	Gießen, RB
6	3	1	17	22	Darmstadt, RB
7	6	1	18	19	Darmstadt
8	6	1	20	21	Frankfurt
4	1	1	24	25	Thüringen

Tabelle 8: Zwischenergebnis nach Schritt 3

id	pid	scope	lft	rgt	name
1	<null>	1	1	26	Bundesrepublik
3	1	1	2	11	Hessen
5	3	1	3	4	Gießen, RB
6	3	1	5	10	Darmstadt, RB
7	6	1	6	7	Darmstadt
8	6	1	8	9	Frankfurt
2	1	1	12	13	Schleswig-Holstein
4	1	1	24	25	Thüringen

Schritt 3: Teilbaum in die Lücke verschieben Nun können wir den Teilbaum in die Lücke verschieben. Allerdings müssen wir zwei mögliche Fälle berücksichtigen: Der Teilbaum wird nach oben verschoben oder nach unten. In unserem Fall wird er nach oben verschoben, das bedeutet, dass wir in Schritt 2 nicht nur eine Lücke geschaffen haben, sondern auch die linken und rechten Positionen des Teilbaums von Hessen verschoben haben. Dies müssen wir beim Verschieben des Bereichs der Zeilen des Teilbaums berücksichtigen.

Wir haben oben bereits `delta` berechnet. Um diesen Wert – im Beispiel `10` – haben wir die hessischen Datensätze nach unten verschoben.

Außerdem müssen wir `shift` berechnen, nämlich wie weit wir den Teilbaum verschieben müssen. Unsere Zielposition ist `target.lft + 1`, also

```
shift = target.lft + 1 - (sub.lft + delta) = 2 - (4 + 10) = -12
```

Folgende SQL-Anweisungen verschieben nun den Bereich zwischen `sub.lft + delta = 4 + 10` und `sub.rgt + delta = 13 + 10` um `shift = -12`:

```
update tree set lft = lft - 12
  where scope = 1 and lft >= 4 + 10 and lft <= 13 + 10;
update tree set rgt = rgt - 12
  where scope = 1 and rgt >= 4 + 10 and rgt <= 13 + 10;
```

Tabelle 8 zeigt das hierdurch entstandene Zwischenergebnis. Und man sieht, dass wir jetzt eine Lücke dort haben, wo der verschobene Teilbaum vorher war.

Schritt 4: Lücke unterhalb schließen Die Lücke ist dort entstanden, wo Hessen vor Schritt 3 war, also müssen wir die Positionen korrigieren, die unterhalb der rechten Position von Hessen in Schritt 2 sind:

```
update tree set lft = lft - 10
  where scope = 1 and lft > 13 + 10;
update tree set rgt = rgt - 10
  where scope = 1 and rgt > 13 + 10;
```

Das Endergebnis wird in Tabelle 9 dargestellt.

Tabelle 9: Endergebnis nach Schritt 4

id	pid	scope	lft	rgt	name
1	<null>	1	1	16	Bundesrepublik
3	1	1	2	11	Hessen
5	3	1	3	4	Gießen, RB
6	3	1	5	10	Darmstadt, RB
7	6	1	6	7	Darmstadt
8	6	1	8	9	Frankfurt
2	1	1	12	13	Schleswig-Holstein
4	1	1	14	15	Thüringen

Pseudocode für das Verschieben eines Teilbaums

Das Vorgehen besteht aus 4 Schritten.

Gegeben sei der Zielknoten `target`, der zu verschiebende Teilbaum mit seiner Wurzel `sub` und die relative Position zum Zielknoten.

Schritt 1: `pid` des Teilbaums ändern

```
-- bestimme die neue pid des Teilbaums
case (zielpos) {
  FIRST_CHILD:   new_pid := target.id
  LAST_CHILD:    new_pid := target.id
  LEFT_SIBLING:  new_pid := target.pid
  RIGHT_SIBLING: new_pid := target.pid
  }
-- trage sie als neue pid des Teilbaums ein
update tree set pid = ?new_pid where id = ?sub.id;
```

Schritt 2: Lücke am Ziel des Verschiebens schaffen Die Position, unter der wir die Lücke schaffen müssen, hängt von der relativen Zielposition am `target` ab. Die Funktion `makeSpace` von oben können wir verwenden, um die Lücke für den Teilbaum zu machen.

```
-- bestimme from: die Position, ab der die Lücke entstehen muss
case (zielpos) {
  FIRST_CHILD:   from := target.lft
  LAST_CHILD:    from := target.rgt - 1
  LEFT_SIBLING:  from := target.lft - 1
  RIGHT_SIBLING: from := target.rgt
  }
-- bestimme delta: die Größe der Lücke
delta = sub.rgt - sub.left + 1
-- bilde die Lücke mit Platz für den zu verschiebenden Teilbaum
makeSpace(from, delta, target.scope)
```

Schritt 3: Teilbaum in die Lücke verschieben Es kann sein, dass die Positionen in unserem Teilbaum `sub` durch das Schaffen der Lücke in Schritt 2 selbst um `delta` verschoben wurden. Dies ist genau dann der Fall, wenn wir den Teilbaum nach oben verschieben. Also müssen wir dies berücksichtigen:

```
-- Bereich des Teilbaums nach Schritt 2
if (sub.lft >= from) {
  range.lft = sub.lft + delta
  range.rgt = sub.rgt + delta
} else {
  range.lft = sub.lft
  range.rgt = sub.rgt
}
-- Wie weit muss er verschoben werden?
shift = from - range.lft
```

Für das Verschieben eines Bereichs machen wir wieder eine Funktion:

```
function moveRange(first, last, shift, scope) {
  update tree set lft = lft + ?shift
    where scope = ?scope and lft between ?first and ?last;
  update tree set rgt = rgt + ?shift
```

```
    where scope = ?scope and rgt between ?first and ?last;
  }
```

Und nun setzen wir diese Funktion ein, um den Teilbaum ans Ziel zu verschieben:

```
moveRange(range.lft, range.rgt, shift, sub.scope)
```

Schritt 4: Lücke unterhalb schließen Im letzten Schritt muss noch die Lücke geschlossen werden, die unterhalb von `range.rgt` durch das Verschieben des Teilbaums entstanden ist. Die Funktion `makeSpace` macht nicht nur eine Lücke, sondern kann sie mit einem negativen `delta` auch wieder schließen:

```
makeSpace(range.rgt, -delta, scope)
```

Löschen von Teilbäumen

Die letzte Aktion zur Manipulation von geordneten Bäumen, die wir betrachten wollen, ist das Löschen eines Teilbaums.

Gegeben sei die Wurzel `sub` des Teilbaums, der gelöscht werden soll.

```
-- Teilbaum sub löschen
-- (1) Knoten des Teilbaums löschen
delete from tree
  where scope = ?sub.scope and lft between ?sub.lft and ?sub.rgt;
-- (2) Entstandene Lücke schließen
makeSpace(sub.rgt, -(sub.rgt - sub.lft + 1), sub.scope)
```

Fazit

Die Analyse hat das Vorgehen bei der Suche und der Manipulation von geordneten Bäumen in SQL dargestellt. Dabei wurde ein hybrides Datenmodell verwendet, das die Baumstruktur durch die Referenz der Knoten auf ihren Vaterknoten beinhaltet und die Ordnung der Knoten durch das Konzept der *nested set*s.

Es hat sich gezeigt, dass Änderungen an der Struktur eines Baums im Konzept der *nested set*s sehr sorgfältig durchgeführt werden müssen, weil in der Regel nicht nur ein Tupel der Tabelle betroffen ist, sondern viele Tupel. Daran zeigt sich eine grundlegende konzeptionelle Schwäche dieses Datenmodells: die Integrität der Baumstruktur kann nur durch Untersuchung der gesamten Tabelle und der Überprüfung der Korrektheit der Angaben zu den linken und rechten Positionen sichergestellt werden.

Nun könnte man einwenden, dass auch beim „natürlichen“ Modell eines Baums in SQL, der ganze Baum betrachtet werden muss, will man etwa feststellen, dass er keinen Zyklus hat oder dass er zusammenhängend ist. Dies ist aber in SQL mittels *Common Table Expressions* leicht möglich.

Hingegen erfordert der Umgang mit *nested set*s in der Praxis eine Bibliothek, die das Konzept sauber implementiert. Solche Bibliotheken gibt es für verschiedene Programmiersprachen. Das bedeutet aber, dass man das Modell eigentlich vernünftig nur im Umfeld eines Zugriffs auf die Daten durch ein Anwendungsprogramm verwenden kann, kaum aber durch direkte Manipulation der Daten in einer SQL-Shell. Schon dieser Sachverhalt zeigt meines Erachtens, dass das Konzept gewissermaßen *windschief* zum relationalen Datenmodell ist.

Deshalb lautet meine Empfehlung, das Konzept der *nested set*s nicht zu verwenden. Aber was tun, wenn man *geordnete* Bäume benötigt? In diesem Fall sollte man sehen, ob nicht die Angaben zu den Knoten selbst eine Ordnung enthalten, die man verwenden kann – eine Ordnung, die sich aus den Angaben selbst ergibt, die in einer Baumstruktur gespeichert werden sollen. In unserem Beispiel mit der politischen Gliederung der Bundesrepublik könnte man den Allgemeinen Regionalschlüssel verwenden, der vom Statistischen Bundesamt vergeben wird und eine geordnete hierarchische Struktur der Gebietskörperschaften der BRD ergibt. Oft sind solche Klassifikationsmerkmale durch das jeweilige Anwendungsgebiet ohnehin vorgegeben.

Literaturverzeichnis

[1] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties.* San Francisco, CA, 2004.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algortihms, 3rd Edition.* 1997.