Effiziente Implementierung einer häufig abgerufenen hierarchischen Struktur in relationalen Datenbanksystemen
†Technischer Bericht – Albert-Ludwigs-Universität Freiburg

Christoph Hermann

Institut für Informatik[†]. Lehrstuhl Algorithmen & DatenStrukturen.
hermann@informatik.uni-freiburg.de

1. April 2008

Zusammenfassung Wir stellen in diesem Bericht eine effiziente Implementierung einer hierarchischen Datenstruktur mittels Nested Sets dar. Eine worst-case Analyse einiger Methoden stellt die Vorteile der Nested Sets, bei der Implementierung häufig abgefragter jedoch selten aktualisierter Datenstrukturen, heraus.

Keywords: Nested set, verschachtelte Hierarchien, Datenbanken, SQL

1 Einleitung

Im electure-Portal der Universität Freiburg werden sämtliche Vorlesungsaufzeichnungen zu Veranstaltungen der Informatik und Mikrosystemtechnik archiviert und teilweise katalogisiert. Zur einfachen Navigation in den Vorlesungsaufzeichnungen sind die Aufzeichnungen für jedes Modul hierarchisch über Ihre zugehörigen Themen gegliedert. Diese Hierarchie besteht aus Kapiteln und Unterkapiteln die beliebig tief geschachtelt sein können. Die Elemente auf derselben Hierarchieebene sind untereinander geordnet. Ähnliche hierarchische Strukturen finden sich in Webforen ("schwarze Bretter") oder hierarchisch geordneten Kommentarlisten (z.B. in Weblogs) wieder.

Alle diese Strukturen haben eines gemeinsam: Sie werden sehr oft abgerufen und deutlich seltener aktualisiert. Im Falle der hierarchischen Struktur zur kapitelweisen Navigation der Aufzeichnungen im electure-Portal ist dieser Unterschied sogar sehr extrem. Die gesamte Struktur wird täglich von den Studierenden in Ihrer Gesamtheit sehr oft abgerufen, sie verändert sich jedoch nur beim Aktualisieren der Inhalte, dem Einstellen neuer Vorlesungsaufzeichnungen. Auch in Webforen und Blogs wird die hierarchische Struktur deutlich häufiger abgefragt als in Ihrer Struktur geändert.

Bei dynamischen Webapplikationen[3] ist die Antwortzeit des Webservers nach der Anfrage eines Benutzers ein kritischer Faktor anhand dessen ein Benutzer enscheidet, ob er diese Applikation weiter verwendet oder nicht. Es ist also ratsam, die Antwortzeit einer Anfrage zu minimieren. Neben weiteren Optimierungsmöglichkeiten wie dem sparsamen Umgang mit Grafiken, der Optimierung des Quellcodes der dargestellten Webseite etc., um die Antwortzeit zwischen Server und Client zu minimieren, ist es auch sinnvoll die Zeit der Generierung der Webseite zu optimieren. Ein Teil dieser

[†] Fakultät für Angerwandte Wissenschaften. Georges-Köhler-Allee 51. D-79110, Freiburg, Deutschland.

[3] Wir gehen hier von Webapplikationen aus, bei denen die Inhalte dynamisch aus einer Datenbank (meist ein relationales Datenbankmanagementsystem) generiert werden.

Zeit wird für Abfragen an die Datenbank, aus der der Inhalt generiert wird, benötigt sowie für die Verarbeitung der gelieferten Ergebnisse.

In diesem Bericht beschäftigen wir uns mit der Frage wie man das Auslesen einer hierarisch in einem relationalen Datenbanksystem gespeicherten Struktur optimieren kann. Hierfür stellen wir im nächsten Abschnitt erstmal ein Modell vor, das es uns erlaubt von der allgemeinen Problematik mit vielerlei Nebenbedingungen zu abstrahieren.

2 Ein allgemeines Abtraktionsmodell

Die in der Einleitung erwähnten hierarchischen Strukturen kann man als Bäume auffassen, wobei die Knoten des Baumes die jeweiligen Kapitel oder Beiträge enthalten. Kapitel und Unterkapitel, bzw. entsprechend Beiträge und darauf bezugnehmende Beiträge in Foren oder Kommentarsystemen werden in der Baumstruktur als miteinander verbundene Knoten dargestellt.

Wir definieren also im Folgenden: Die Hierarchie bildet einen Baum T mit Knoten V und Kanten E mit $E \subset V \times V$; $T = (V, E)$ mit folgenden Eigenschaften:

- Es gibt einen Knoten $r \in V$, die Wurzel des Baumes. Die Wurzel des Baumes hat keine Vorfahren.
- Knoten ohne Nachfolger sind Blätter des Baumes.
- Jeder Knoten $x \in V$ wird über eine eindeutige Zahl (id) identifiziert, die einmalig im gesamten Baum ist.
- Als *level* eines Knoten bezeichnen wir die Länge des Pfades von der Wurzel r bis zum aktuellen Knoten.

3 Definitionen

Im Folgenden beziehen wir unsere Beschreibungen auf den bereits beschriebenen Fall des electure-Portals, für die anderen Varianten hierarischer Strukturen gelten diese weitestgehend analog. Die verwendete Implementierung der Hierarchie muss Funktionalitäten zum Abfragen der Hierarchie, sowie zur Manipulation (Löschungen, Einfügungen, Verschiebungen) dieser bereitstellen.

In Celko (2004) werden vier Varianten von Hierarchien vorgestellt: Hierarchien mit statischen Knoten und statischen Kanten, mit statischen Knoten und dynamischen Kanten, mit dynamischen Knoten und statischen Kanten und Hierarchien mit dynamischen Knoten und dynamischen Kanten.

Die von uns verwendeten Hierarchie ist eine des Typs statische Knoten und dynamische Kanten. Der Inhalt der Knoten ändert sich in der Regel nicht, es ist jedoch möglich neue Knoten hinzuzufügen. Des Weiteren stellt das Abfragen der gesamten Hierarchie die häufigste Nutzungsart in unserem Fall dar. Manipulationen kommen im Vergleich dazu relativ selten vor.

Folgende Anforderungen sind demnach von einer Datenstruktur zur Abbildung dieser Hierarchie zu erfüllen:

Strukturerhaltend
: Die Hierarchie muss strukturerhaltend abgespeichert werden. Hierarchische Relationen müssen entsprechend in der Datenstruktur abgebildet werden.

Unbegrenzte Verschachtelungstiefe
: Es darf keinerlei Einschränkungen bezüglich der Verschachtelungstiefe geben.

Sortierbarkeit
: Die Reihenfolge der Elemente muss definierbar sein.

Funktionalität
: Es müssen Methoden zur Abfrage der (gesamten und teilweisen) Datenstruktur, sowie zur Manipulation (Löschen, Einfügen, Verschieben von Teilstrukturen) derselbigen zur Verfügung stehen.

Geschwindigkeit
Die Abfrage der Datenstruktur muss besonders effizient erfolgen, da dies die häufigste Funktionalität darstellt.

4 Mögliche Implementierungen hierarchischer Strukturen in relationalen Datenbanken

Zur Implementierung des definierten Modells stehen verschiedene Möglichkeiten zur Verfügung: Unter anderem das Adjazenzlistenmodell, das Pfad-Modell sowie das Nested Set Modell.

4.1 Das Adjanzenlistenmodell

Das Adjazenlistenmodell ist das klassische Modell, zur Darstellung von Baumstrukturen in relationalen Datenbanken. Hierbei wird in jedem Knoten eine Referenz $r(x)$ des zughörige Vaterknotens abgespeichert. Sei $G = (V, E)$ ein Graph und $\{x_1, x_2\} \in E$. Dann stellt $A(x_1) = \{x_2 \in V : \{x_1, x_2\} \in E\}$ die Relation von x_1 und x_2 dar. Der Wurzelknoten hat keinen Vaterknoten, die Referenz wird hier also auf *null* gesetzt $r(x) = null$.

Das Hauptproblem des Adjazenzlistenmodells stellt die Rekursion zur Abfrage der Datenstruktur dar. In einem ersten Schritt muss der Wurzelknoten ermittelt werden ($r(x) == null$), und danach rekursiv für jeden Knoten x die Liste seiner Nachfolger: $nachfolger(x_1) = \{x \in V : r(x) == x_1\}$. Diese Art von Abfrage ist mit einem relationalen Datenbankmanagementsystem nur sehr ineffizient auszuführen.

4.2 Das Pfadmodell

Das Pfad-Modell (hierbei wird bei jedem Knoten der Pfad von der Wurzel zum aktuellen Knoten abgespeichert) verletzt die Bedingung der *unbegrenzten Verschachtelungstiefe*[4]. Deshalb wird sowohl dieses als auch das Adjazenzlisten-Modell im Folgenden nicht weiter betrachtet.

Das Nested Set Modell, das im folgenden Abschnitt vorgestellt wird, erfüllt alle unsere Bedingungen aus Abschnitt 3.

4.3 Das Nested Set Modell

Im Nested Set Modell wird die hierarchische Baumstruktur durch ineinander verschachtelte Mengen repräsentiert. Jeder Knoten x des Baums wird als Menge M dargestellt, wobei die Nachfolger eines Knotens in der Menge des Vaterknotens enhalten sind. Eine Menge M_2 ist also in M_1 enthalten, wenn die zugehörigen Knoten x_1, x_2 miteinander über einen Pfad verbunden sind, so dass x_2 ein Nachfahre von x_1 ist. Der Wurzelknoten wird also als Menge repräsentiert die alle anderen Mengen enthält. In Abbildung 1 stellen wir diese Zusammenhänge dar.

[4] Obwohl die tatsächliche Grenze in der Praxis wohl nur sehr selten erreicht werden würde.

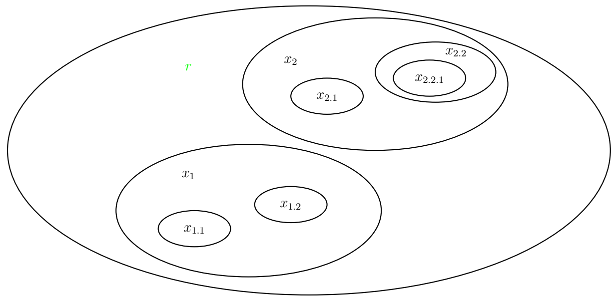

Abbildung 1. Darstellung eines Baums im Nested Set Modell

Diese Mengen und Ihre Relationen sollen nun in einer relationalen Datenbank gespeichert werden. Zur Herleitung der entsprechenden Abbildung auf n-Tupel in der Datenbank bedienen wir uns der Darstellung derselben Mengen mittels verschachtelter Intervalle. In Abbildung 2 zeigen wir die Darstellung des Nested Sets aus Abbildung 1 als verschachtelte Intervalle.

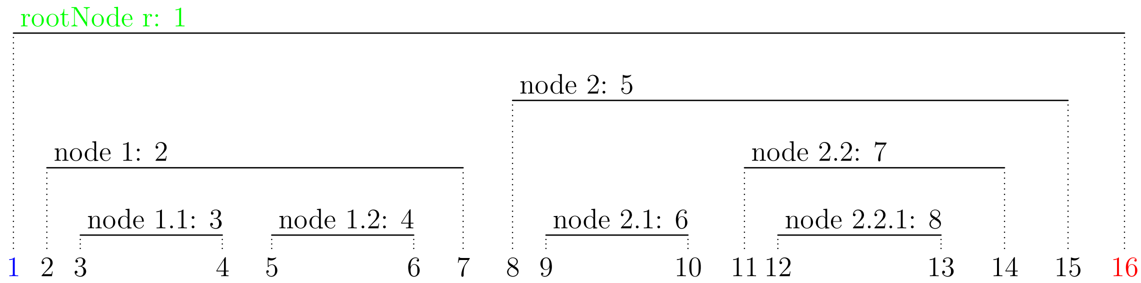

Abbildung 2. Darstellung des Nested Set als verschachtelte Intervalle

Wie aus Abbildung 2 ersichtlich ist jede Menge durch zwei Zahlen eindeutig identifiziert. Wir bezeichnen im Folgenden die Zahl die dem Intervallanfang zugeordnet ist als *leftId* und die Zahl die dem Intervallende zugeordnet wird als *rightId*. Anhand dieser Darstellungsform kann man einige Eigenschaften der Nested Sets identifizieren:

- Die *leftId* einer Menge ist immer kleiner als die leftId aller enthaltenen Mengen und die *rightId* ist immer größer. Daraus folgt, dass die *leftId* und *rightId* einer Menge immer zwischen denen der Obermenge ist.
- Für alle Mengen ohne Untermengen gilt *rightId*-*leftId*=1.
- Die Anzahl der Untermengen einer Menge ist immer (*rightId*-*leftId*-1)/2.
- Die Knoten auf dem Pfad von einer Obermenge zu einer Teilmenge sind immer bezüglich der *leftId* aufsteigend sortiert.

Überführt man nun die Darstellung der Mengen als verschachtelte Intervalle in eine Baumstruktur, wobei jeder Knoten zusätzlich zu den *leftId* und *rightId* Werten eine eindeutige Id erhält, so erhält man die Darstellung in Abbildung 3.

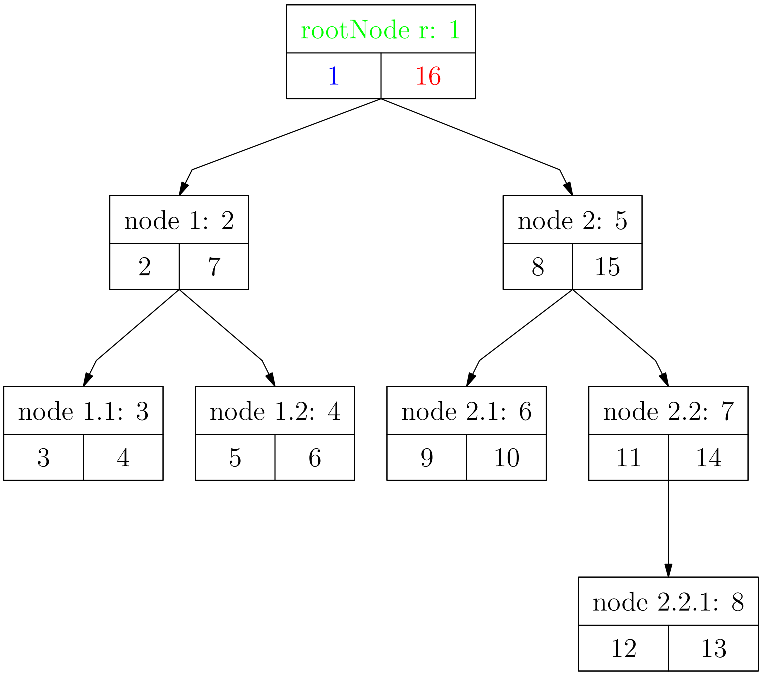

Abbildung 3. Darstellung des Nested Set als Baumstruktur mit *leftId*s und *rightId*s

Diese Baumstruktur kann nun so in SQL übertragen werden. Jeder Knoten des Baumes wird als Tupel Nestedset(id, title, *leftId*, *rightId*, *rootId*) dargestellt. Zusätzlich zu den bisher bekannten Werten (id, leftId, rightId) werden zwei weitere Werte title und *rootId* eingeführt. title ist lediglich ein String (Name des entsprechenden Knotens) der keine weitere Bedeutung für die Baumstruktur hat[5]. *rootId* entspricht der id des Wurzelknotens. Das hat zwei Vorteile: Zum einen können so in einer Tabelle in SQL mehrere Bäume abgespeichert werden (jeder Baum besteht aus den Tupeln mit derselben rootId). Selbstverständlich wäre es möglich jeden anderen Wert zu verwenden, um einen Baum zu identifizieren. Zum anderen hat die Verwendung der id des Wurzelknotens als *rootId* den Vorteil, dass man von jedem Knoten aus direkt zum Wurzelknoten kommen kann, die *rootId* dient also als Referenz auf den Wurzelknoten. Zusätzlich könnte man in jedem Knoten noch eine Referenz auf den Vorgängerknoten abspeichern, dies ist in unserem Szenario aber nicht notwendig und kann auch über die *leftId* und *rightId* Werte berechnet werden.

Um die BaumStruktur abzurufen und manipulieren zu können benötigt man eine Reihe von Methoden, die im Folgenden definiert werden:

appendNodeInside(nestedsetNode ref, String title) Diese Methode fügt dem Baum einen neuen Knoten als rechtesten Nachfolger des Referenzknotens (ref) hinzu.

appendNodeLeft(nestedsetNode ref, String title) Diese Methode fügt einen Knoten als linken Nachbarn des Referenzknotens ref ein.

appendNodeRight(nestedsetNode ref, String title) Analog zu *appendNodeLeft* wird hier ein neuer Knoten als rechter Nachbar des Referenzknotens eingefügt.

deleteNode(nestedsetNode ref) Diese Methode löscht den Referenzknoten.

deleteSubtree(nestedsetNode ref Diese Methode löscht den Teilbaum, dessen Wurzel der Referenzknoten ist. Prinzipiell wird bei *deleteNode* genau wie bei *deleteSubtree* ein Teilbaum gelöscht,

[5] Dieser Wert wird bei uns der Kapitelbezeichnung entsprechen.

wir nehmen diese Unterscheidung aus Performance-Gründen vor, wie bei der Implementierung im Folgenden gezeigt werden wird.

moveSubtreeInside(nestedsetNode ref, nestedsetNode end) Diese Methode verschiebt einen Teilbaum mit Wurzel ref und hängt Ihn unterhalb des Knotens end wieder ein.

moveSubtreeLeft(nestedsetNode ref, nestedsetNode end) Diese Methode verschiebt einen Teilbaum mit Wurzel ref, so dass ref zum linken Nachbarn des Knotens end wird.

moveSubtreeRight(nestedsetNode ref, nestedsetNode end) Diese Methode verschiebt einen Teilbaum analog zu *moveSubtreeLeft*, so dass ref zum rechten Nachbarn des Knotens end wird.

moveLeafNodeInside(nestedsetNode ref, nestedsetNode end),
moveLeafNodeLeft(nestedsetNode ref, nestedsetNode end),
moveLeafNodeRight(nestedsetNode ref, nestedsetNode end) Diese Methoden implementieren dieselbe Funktionalität wie die entsprechenden moveSubtree* Methoden, mit dem Unterschied, dass jeweils nur ein Blatt anstatt eines Teilbaumes verschoben wird. Dies hat wie schon bei den delete* Methoden Performance-Gründe.

Die Implementierung in SQL all dieser Methoden findet sich im Anhang dieses Berichts.

Im Folgenden wollen wir nur kurz nocheinmal anhand ein paar Abbildungen die Konstruktion bzw. das Einfügen mehrerer Knoten in den Baum nachvollziehen:

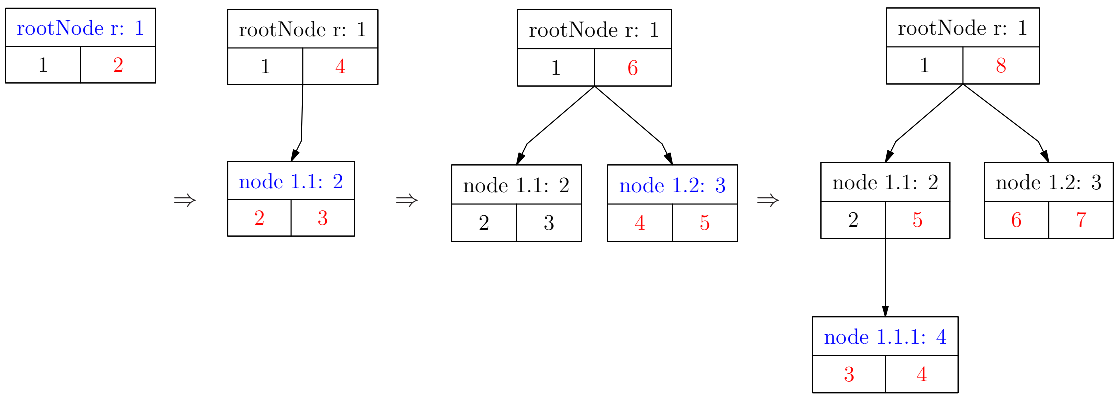

Abbildung 4. Konstruktion des Baumes

Rot hervorgehoben sind jeweils die geänderten Werte nach dem Einfügen des neuen Knotens (blau hervorgehoben). Man sieht hier auch schon sehr schön, dass ein Einfügen neuer Knoten, die Modifikation der Werte aller Knoten nach sich ziehen kann. Wie man an den Worst-Case Analysen im nächsten Abschnitt sehen wird, sind Löschungen relativ einfach durchzuführen, führen aber genau wie Inserts im Worst-Case zum Update aller Tupel des Baumes in SQL. Daniel Brandon beschreibt in Brandon (2005) genau dieses Problem, schlägt aber keinerlei Problemlösung hierfür vor. Eine mögliche Problemlösung ist das schrittweise Verändern der *leftId*s und *rightId*s der Knoten. Wählt man eine grössere Schrittweite als 1, so können mehrere Knoten in den Baum eingefügt werden, ohne dass alle Knoten mit größerer *leftId* bzw. *rightId* verändert werden müssen. Über die Wahl der Schrittweite kann somit die Updatehäufigkeit des kompletten Baumes minimiert werden. Daniel Brandon beschreibt auch das Problem, dass es sehr kostspielig ist. einen bottom-up Durchlauf des Baumes durchzuführen. Eine mögliche Lösung hierfür wäre das zusätzliche Abspeichern des jeweiligen Vaterknotes, was jedoch wiederum zu einer rekursiven Abfrage der Datenbank führen würde. Es ist also deutlich einfacher den Weg vom Wurzelknoten zu einem Nachfolgeknoten zu nehmen als umgekehrt. Dieser Ansatz wird von uns durch die Einführung der *rootId* ermöglicht (Siehe Seite 5).

4.4 Worst-Case Analyse des Nested Set Modells

Wie im Abschnitt 4.3 beschrieben, wird die Baumstruktur als n-Tupel Nestedset(id, title, *leftId*, *rightId*, *rootId*) dargestellt. Im Code-Ausschnitt 1 wird die CREATE-TABLE-Anweisung zur Erstellung der entsprechenden Tabelle dargestellt.

Code-Ausschnitt 1 Struktur der Nestedset Tabelle in SQL

```sql
CREATE TABLE Nestedset (
  id NUMBER NOT NULL,
  title VARCHAR(255),
  rootId NUMBER REFERENCES Nestedset(id),
  leftId NUMBER,
  rightId NUMBER,
  PRIMARY KEY (id)
);
```

Im Folgenden wollen wir exemplarisch anhand der Methoden *appendNodeInside* und *deleteSubtree* eine Worst-Case Analyse durchführen. Wir nehmen im Folgenden an, dass drei Indices auf der Tabelle Nestedset definiert sind: *index(leftId)*, *index(rightId)* und *index(rootId)* die ensprechend Ihrem Zweck entsprechend effizient sind (auf die Indices auf *rightId* und *leftId* wird überwiegend mit Vergleichsoperationen wie >=,<=,< und > zugegriffen (mit mysql kann hier z.B. ein BTREE Index verwendet werden), während für den Index auf der *rootId* überwiegend für den direkten Zugriff auf Elemente mit diesem Wert verwendet wird (rootId = $wert; hier sollte z.B. ein HASH Index verwendet werden).

appendNodeInside

Code-Ausschnitt 2 appendNodeInside in SQL

```sql
UPDATE Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId > $leftId;
UPDATE Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId >= $rightId;
INSERT INTO Nestedset (title,rootId,leftId,rightId)
  VALUES ('title',$rootId,$rightId,$rightId + 1);
```

Code-Ausschnitt 2 zeigt die SQL-Anweisungen zum Einfügen eines neuen Knotens. *$rootId*, *$leftId* und *$rightId* stehen hier für die Werte des Referenzknotens unter dem der neue Knoten eingefügt wird. Unter der Annahme, dass das Auffinden der zu ändernden Tupel durch die definierten Indices performanter ist, als das eigentliche Ändern der Werte der betroffenen Tupel, entspricht das

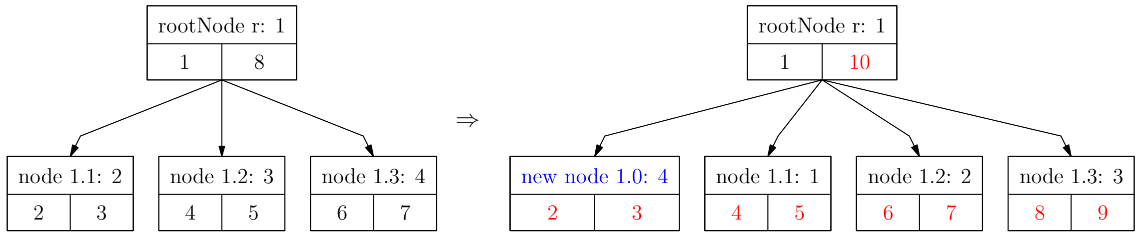

Abbildung 5. Worst Case für das Einfügen eines neuen Knotens

Einfügen eines Knotens ganz links unterhalb des Wurzelknotens dem Worst-case, da für alle Knoten mindestens die *rightId* aktualisiert werden muss. Abbildung 5 veranschaulicht dies.

Da alle n Tupel des Baumes ($|V| = n$) aktualisiert werden müssen entsprechen die Kosten der Einfügung eines Knotens der folgenden Gleichung (f = Menge der Tupel nach Ausführung eines Filters (Where-Bedingung), u = Menge der betroffenen Tupel nach Ausführen aller Filter):

$$\begin{aligned} c_{appendNodeInside} &= c(f_{index(rootId=\$rootId)}) + c(f_{index(leftId>\$leftId)}) + c_{updateLeftId}(u_1) \\ &+ c(f_{index(rootId=\$rootId)}) + c(f_{index(rightId>=\$rightId)}) + c_{updateRightId}(u_2) \\ &+ c_{insert} \end{aligned}$$

Lässt man das Caching der Datenbank ausser Betracht[6] lässt sich diese Gleichung reduzieren auf:

$$\begin{aligned} c_{appendNodeInside} &= 2 * c(f_{index(rootId=\$rootId)}) + c(f_{index(leftId>\$leftId)}) + c_{updateLeftId}(u_1) \\ &+ c(f_{index(rightId>=\$rightId)}) + c_{updateRightId}(u_2) \\ &+ c_{insert} \end{aligned}$$

Bei einer reinen Betrachtung eines einzigen Nested Sets in der Datenbank (die Filterbedingung auf die *rootId* kann entfallen[7]) und der Annahme, dass die Filterbedingungen sowie das Einfügen eines neuen Tupels annähernd konstante Zeit $O(1)$ benötigen) kommen wir zu der Gleichung:

$$\begin{aligned} c_{appendNodeInside} &= 2 * O(1) + c_{updateLeftId}(u_1) \\ &+ c_{updateRightId}(u_2) \\ &+ O(1) \\ &= O(1) + c_{updateLeftId}(u_1) + c_{updateRightId}(u_2) \end{aligned}$$

Da die zwei Updates einmal alle Tupel des Baumes und einmal alle ausgenommen der Wurzel betreffen erhalten wir einen worst case von $O(n)$:

$$\begin{aligned} c_{appendNodeInside} &= O(1) + c(2 * n - 1) \\ &= O(n) \end{aligned}$$

Das lässt sich logisch auch ganz einfach nachvollziehen, da für alle Knoten ausser dem Wurzelknoten die *leftId* und *rightId* geändert werden muss und für den Wurzelknoten die *rightId*.

---

[6] Mit Caching ist das zweite Ausführen der Filterbedingung $rootId = \$rootId$ annähernd konstant

[7] Entfällt diese Filterbedingung nicht, erhalten wir $5 * k$ anstatt $3 * k$, was nichts an der Komplexität von $O(n)$ ändert.

deleteSubTree

Soll ein kompletter Teilbaum gelöscht werden, so entsteht im restlichen Baum ein „Loch“, d.h. die *leftIds* und *rightIds* der nachfolgenden Knoten haben nicht die richtigen Werte und müssen aktualisiert werden. Die *leftIds* und *rightIds* der Nachfolgerknoten müssen jeweils um den Wert (rightId - leftId + 1) verringert werden, da dieser Wert genau der zweifachen Anzahl der Knoten des Teilbaumes inklusive seiner Wurzel entspricht ($|V| = (rightId - leftId + 1)/2$; Diese Eigenschaft wurde im Abschnitt 4.3 als Eigenschaften der Nested Sets vorgestellt).

Code-Ausschnitt 3 deleteSubTree in SQL

```sql
DELETE FROM Nestedset
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - ($rightId - $leftId + 1)
  WHERE leftId > $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET rightId = rightId - ($rightId - $leftId + 1)
  WHERE rightId > $rightId
    AND rootId = $rootId;
```

Dies führt zu folgender Worst-Case Analyse für *deleteSubTree*:

$$\begin{aligned} c_{deleteSubTree} = {} & c(f_{index(leftId>=\$leftId)}) + c(f_{index(rightId<=\$rightId)}) + c(f_{index(rootId=\$rootId)}) + c_{delete}(u_1) \\ & + c(f_{index(leftId>\$leftId)}) + c(f_{index(rootId=\$rootId)}) + c_{updateLeftId}(u_2) \\ & + c(f_{index(rightId>\$rightId)}) + c(f_{index(rootId=\$rootId)}) + c_{updateRightId}(u_3) \end{aligned}$$

Dies lässt sich wiederum reduzieren auf:

$$\begin{aligned} c_{deleteSubTree} = {} & 4 * O(1) + c_{delete}(u_1) \\ & + c_{updateLeftId}(u_2) \\ & + c_{updateRightId}(u_3) \end{aligned}$$

Mit $t = (V_t, E_t)$, $n_t = |V_t|$ als zu löschendem Teilbaum mit Wurzelknoten r_t erhalten wir somit:

$$\begin{aligned} c_{deleteSubTree} &= 4 * O(1) + c_{delete}(|V_t|) \\ & \quad + c_{updateLeftId}(|V \setminus V_t|) \\ & \quad + c_{updateRightId}(|V \setminus V_t|) \\ &= O(1) + c_{delete}(|V_t|) + 2 * c_{update}(|V \setminus V_t|) \end{aligned}$$

Geht man wiederum davon aus, dass sowohl das Löschen eines Tupels, als auch das aktualisieren eines Tupels annähernd dieselbe Zeit in Anspruch nehmen, so erhält man:

$$\begin{aligned} c_{deleteSubTree} &= O(1) + c(2 * |V| - |V_t|) \\ &= O(1) + c(2 * n - n_t) \\ &= O(n) \end{aligned}$$

was wiederum in einer linearen Komplexität $O(n)$ resultiert. Der Unterschied zwischen dem Löschen eines Teilbaumes und dem Löschen eines Blattes besteht in der Anzahl der Indices, auf die die Datenbank zurückgreifen muss. Daher kann das Löschen eines Blattes mit anderen SQL-Anweisungen performanter ausgeführt werden.

Abfragen aller Tupel

Code-Ausschnitt 4 Abfragen aller Knoten in SQL

```sql
SELECT FROM Nestedset
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId
  ORDER BY leftId;
```

Das Abfragen aller Tupel der entsprechenden Knoten eines Baumes geschieht in annähernd konstanter Zeit[8]:

$$\begin{aligned} c_{query} &= c(f_{index(leftId>=\$leftId)}) + c(f_{index(rightId<=\$rightId)}) + c(f_{index(rootId=\$rootId)}) \\ &= 3 * O(1) \end{aligned}$$

5 Derzeitiger Stand, Optimierungsmöglichkeiten und Ausblick

Derzeitig liegt eine vollständige Implementierung des Nested Sets implementiert mit Java (Hibernate) und SQL für das electure-Portal vor. Bisher wurde noch nicht in einem echten Szenario erprobt inwiefern die Implementierung mittels eines Nested Sets tatsächlich bessere Performance zeigt als eine gewöhnliche Implementierung mit dem Adjazenzmodell oder einem vergleichbaren Modell. Des Weiteren bleibt zu ermitteln, inwiefern weiteres Optimierungspotential bzgl. Caching innerhalb des Web Application Servers oder Optimierung der Indices ausgeschöpft werden kann.

In diesem Bericht haben wir gezeigt, warum eine Implementierung einer häufig abgefragten Hierarchie mittels Nested Sets bessere Eigenschaften für diesen Fall hat als gewöhnliche Implementierungen mit dem Adjazenz- oder dem Pfadmodell.

Ausführliche (Praxis-)Tests sowie die genauere Untersuchung der alternativen Modelle stehen bisher noch aus. Insbesondere der Optimierung der Indices sollte ein besonderes Augenmerk geschenkt werden. In Abschnitt 4.4 haben wir kurz angesprochen, dass für mySQL ein HASH-Index auf die *rootId* sowie BTREE Indices auf *leftId* und *rightId* empfehlenswert sind, da diese für den jeweiligen Einsatzzweck besonders geeignet erscheinen. Es bleibt offen, ob andere Datenbanksysteme wie z.B. von Oracle o.ä. noch effizientere Strukturen für die Abbildung einer hierarchischen Struktur implementiert haben.

[8] Natürlich ist diese Abfrage auch abhängig von der Größe der zurückgelieferten Menge und der tatsächlichen Dauer der Abfrage der Indizes. Auch der Ausführungszeit der Sortierung wird hier keine weitere Beachtung geschenkt, da diese für das Abfragen der Knoten nicht relevant ist.

Literatur

[Brandon 2005] Brandon, Daniel: Recursive database structures. In: *Journal of Computing Sciences in Colleges* 21 (2005), Nr. 2, S. 295–304. – ISSN 1937-4771

[Celko 2004] Celko, John: *Trees and Hierarchies in SQL for Smarties*. Elsevier, 2004. – ISBN 1-55860-920-2

A SQL-Implementierung der Methoden zum Abrufen und Manipulieren von Nested Sets in relationalen Datenbanken

A.1 fetch

Code-Ausschnitt 5 Abfragen aller Knoten in SQL

```sql
SELECT FROM Nestedset
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
```

A.2 fetchWithLevel

Code-Ausschnitt 6 fetchWithLevel in SQL

```sql
SELECT ns.nestedsetId, ns.rootId, ns.rightId, ns.leftId, ns.title, COUNT(*) - 1 AS
    level
   FROM Nestedset ns, Nestedset ns2
  WHERE ns.rootId = $rootId
    AND ns2.rootId = $rootId
    AND ns.leftId >= ns2.leftId
    AND ns.leftId <= ns2.rightId
  GROUP BY ns.leftId
  ORDER BY ns.leftId
```

A.3 appendNodeInside

Code-Ausschnitt 7 appendNodeInside in SQL

```
UPDATE  Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
UPDATE  Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId >= $rightId;
INTO Nestedset (title,rootId,leftId,rightId)
  VALUES ('title',$rootId,$rightId,$rightId + 1);
```

A.4 appendNodeLeft

Code-Ausschnitt 8 appendNodeLeft in SQL

```
UPDATE  Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId >= $leftId;
UPDATE  Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId > $leftId;
INTO Nestedset (title,rootId,leftId,rightId)
  VALUES ('title',$rootId,$leftId,$leftId + 1);
```

A.5 appendNodeRight

Code-Ausschnitt 9 appendNodeRight in SQL

```
UPDATE Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId > $rightId;
UPDATE  Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
INTO Nestedset (title,rootId,leftId,rightId)
  VALUES ('title',$rootId,$rightId + 1,$rightId + 2);
```

A.6 deleteLeafNode

Code-Ausschnitt 10 deleteLeafNode in SQL

```sql
DELETE FROM Nestedset
  WHERE id = $id
    AND rootId = $rootId;
UPDATE  Nestedset
    SET leftId = leftId - 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
UPDATE  Nestedset
    SET rightId = rightId - 2
  WHERE rootId = $rootId
    AND rightId > $rightId;
```

A.7 deleteSubTree

Code-Ausschnitt 11 deleteSubTree in SQL

```sql
DELETE FROM Nestedset
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - $change
  WHERE leftId > $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET rightId = rightId - $change
  WHERE rightId > $rightId
    AND rootId = $rootId;
```

A.8 moveLeafNodeInside

Code-Ausschnitt 12 moveLeafNodeInside in SQL

```
UPDATE Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId = $leftId
    AND rightId = $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
UPDATE Nestedset
    SET rightId = rightId - 2
  WHERE rootId = $rootId
    AND rightId > $rightId;
UPDATE Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId >= $newparentrightId;
UPDATE Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId > $newparentrightId;
UPDATE Nestedset
    SET leftId = $newparentrightId, rightId = $newparentrightId + 1
  WHERE rootId = $rootId
    AND nestedsetId = $nestedsetToMoveId;
```

A.9 moveLeafNodeLeft

Code-Ausschnitt 13 moveLeafNodeLeft in SQL

```sql
UPDATE Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId = $leftId
    AND rightId = $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
UPDATE Nestedset
    SET rightId = rightId - 2
  WHERE rootId = $rootId
    AND rightId > $rightId;
UPDATE Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId >= $brotherleftId;
UPDATE Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId > $brotherleftId;
UPDATE Nestedset
    SET leftId = $brotherleftId, rightId = $brotherleftId + 1
  WHERE rootId = $rootId
    AND nestedsetId = $nestedsetToMoveId;
```

A.10 moveLeafNodeRight

Code-Ausschnitt 14 moveLeafNodeRight in SQL

```sql
UPDATE Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId = $leftId
    AND rightId = $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - 2
  WHERE rootId = $rootId
    AND leftId > $rightId;
UPDATE Nestedset
    SET rightId = rightId - 2
  WHERE rootId = $rootId
    AND rightId > $rightId;
SELECT Nestedset.* FROM Nestedset
  WHERE rootId = $rootId;
    AND nestedsetId = $nestedsetBrotherId;
UPDATE Nestedset
    SET leftId = leftId + 2
  WHERE rootId = $rootId
    AND leftId > $brotherrightId;
UPDATE Nestedset
    SET rightId = rightId + 2
  WHERE rootId = $rootId
    AND rightId > $brotherrightId;
UPDATE Nestedset
    SET leftId = $brotherrightId + 1, rightId = $brotherrightId + 2
  WHERE rootId = $rootId
    AND nestedsetId = $nestedsetToMoveId;
```

A.11 moveSubtreeInside

Code-Ausschnitt 15 moveSubtreeInside in SQL

```
UPDATE Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - $change
  WHERE leftId > $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET rightId = rightId - $change
  WHERE rightId > $rightId
    AND rootId = $rootId;
SELECT Nestedset FROM Nestedset
  WHERE rootId = $rootId
    AND nestedsetId = $nestedsetParentId;
UPDATE Nestedset
    SET leftId = leftId + $change
  WHERE rootId = $rootId
    AND leftId > $newparentrightId;
UPDATE Nestedset
    SET rightId = rightId + $change
  WHERE rootId = $rootId
    AND rightId >= $newparentrightId;
UPDATE Nestedset
    SET leftId = (leftId - $newparentleftId + $leftId - 1) * -1, rightId = (rightId -
         $newparentleftId + $leftId - 1) * -1
  WHERE rootId = $rootId
    AND (leftId < 0 OR rightId < 0);
```

A.12 moveSubTreeLeft

Code-Ausschnitt 16 moveSubTreeLeft in SQL

```
UPDATE  Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
UPDATE  Nestedset
    SET rightId = rightId - $change
  WHERE rightId > $rightId
    AND rootId = $rootId;
UPDATE  Nestedset
    SET leftId = leftId - $change
  WHERE leftId > $rightId
    AND rootId = $rootId;
SELECT Nestedset.* FROM Nestedset
  WHERE rootId = $rootId
    AND nestedsetId = nestedsetBrotherId;
UPDATE  Nestedset
    SET leftId = leftId + $change
  WHERE rootId = $rootId
    AND leftId >= $brotherleftId;
UPDATE  Nestedset
    SET rightId = rightId + $change
  WHERE rootId = $rootId
    AND rightId > $brotherleftId;
UPDATE  Nestedset
    SET leftId = (leftId + leftId - $brotherleftId) * -1, rightId = (rightId +
        $leftId - $brotherleftId) * -1
  WHERE rootId = $rootId
    AND (leftId < 0 OR rightId < 0);
```

A.13 moveSubTreeRight

Code-Ausschnitt 17 moveSubTreeRight in SQL

```sql
UPDATE Nestedset
    SET leftId = leftId * -1, rightId = rightId * -1
  WHERE leftId >= $leftId
    AND rightId <= $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET rightId = rightId - $change
  WHERE rightId > $rightId
    AND rootId = $rootId;
UPDATE Nestedset
    SET leftId = leftId - $change
  WHERE leftId > $rightId
    AND rootId = $rootId;
SELECT Nestedset FROM Nestedset
  WHERE rootId = $rootId
    AND nestedsetId = $nestedsetBrotherId;
UPDATE Nestedset
    SET leftId = leftId + $change
  WHERE rootId = $rootId
    AND leftId > $brotherrightId;
UPDATE Nestedset
    SET rightId = rightId + $change
  WHERE rootId = $rootId
    AND rightId > $brotherrightId;
UPDATE  Nestedset
    SET leftId = (leftId + $leftId - $brotherrightId - 1) * -1, rightId = (rightId +
        $leftId - $brotherrightId -1) * -1
  WHERE rootId = $rootId
    AND (leftId < 0 OR rightId < 0);
```