Eine effiziente Indexstruktur für dynamische hierarchische Daten

Robert Brunel · Jan Finis

Technische Universität München
Institut für Informatik
Lst. III: Datenbanksysteme
Boltzmannstr. 3
85748 Garching
vorname.nachname@in.tum.de

Abstract: Bis heute fällt es relationalen Datenbanksystemen schwer, große Mengen dynamischer hierarchischer Daten effizient zu verwalten, obwohl dies eine ständig wiederkehrende Anforderung in fast allen betrieblichen Informationssystemen ist. In dieser Arbeit stellen wir eine Datenstruktur vor, die beliebig geformte Hierarchien derart indexiert, dass strukturbezogene Anfragen beschleunigt werden, während gleichzeitig strukturelle Änderungen wie das Einfügen und Löschen von Knoten in logarithmischer Zeit möglich sind. Wir verfolgen damit langfristig das Ziel, in relationalen Datenbanken künftig Hierarchien als native, den Relationen praktisch gleichgestellte Datenobjekte zu unterstützen.

1 Einleitung

Das Ablegen hierarchischer Daten gehört zu den ältesten Anforderungen von Geschäftsanwendungen an Datenbanksysteme überhaupt. Betriebliche Informationssysteme nutzen Hierarchien etwa zur Verwaltung personeller und geografischer Organisationsstrukturen, in Projektplanungsprozessen, zur hierarchischen Untergliederung von Bauplänen (so genannte Bills of Materials), oder auch im Bereich der Business Intelligence, wo man die zu analysierenden Geschäftsdaten mittels „Slice and Dice“ entlang so genannter Dimensionshierarchien filtrieren und auf die gewünschte Detailstufe reduzieren möchte. Seit relationale Datenbanksysteme (RDBMS) die hierarchischen Datenbanken wie IBM IMS weitestgehend abgelöst haben, sind sie dafür kritisiert worden, hierarchische Daten nicht adäquat abbilden zu können. Diese Tatsache ist auf die inhärent flache Natur des relationalen Modells zurückzuführen – ein klassischer „Impedance Mismatch“. In den vergangenen 35 Jahren hat sich daran erstaunlich wenig geändert: Während objektorientierte, XML-, Graph- und mehrdimensionale Datenbanksysteme gewisse Formen von Hierarchien als primäre Datenobjekte nativ unterstützen, stehen den Benutzern handelsüblicher RDBMS wenn überhaupt nur sehr beschränkte SQL-Erweiterungen zu Verfügung – beispielsweise rekursive `WITH`-Ausdrücke oder das `CONNECT-BY`-Konstrukt [Hau11, Ora10] –, die weder den benötigten Funktionsumfang abdecken noch eine für uns akzeptable Performanz haben.

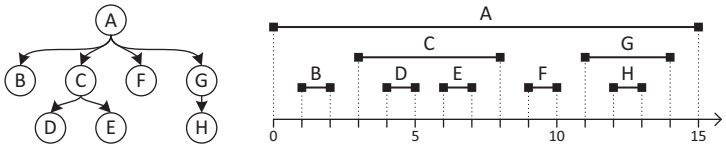

Abbildung 1: Eine Beispielhierarchie und ihre Kodierung durch geschachtelte Intervalle.

Im Rahmen unseres Forschungsprojektes suchen wir nach Möglichkeiten, relationale Datenbanksysteme um spezielle Indexstrukturen für Hierarchien so zu erweitern, dass sie die typischen Anwendungsfälle von Geschäftsanwendungen weitestgehend abdecken. Zum einen konzentrieren wir uns dabei auf sehr *große* Hierarchien mit Knotenzahlen im zweistelligen Millionenbereich oder noch höher. Auch degenerierte, z. B. sehr flache und breite oder sehr tiefe Strukturen sollen die Anfrageauswertung möglichst nicht beeinträchtigen. Zum anderen gehen wir von stark *dynamischen* Datensätzen aus. Die wesentlichen auf Hierarchien denkbaren Änderungsoperationen „Knoten einfügen“, „Knoten entfernen“ und „Teilbaum verschieben“ möchten wir in hohen Raten von mindestens einigen zehntausend pro Sekunde unterstützen. Anders als manche vergleichbare Ansätze zur Indexierung von XML-Daten (vgl. Abschnitt 5) bauen wir dabei nicht auf dem relationalen Schema auf, sondern streben eine Erweiterung des Datenbankkerns an, was uns zusätzlichen Spielraum gibt, den Speicherplatzverbrauch zu minimieren und gleichzeitig die Performanz zu maximieren.

Im Folgenden stellen wir eine in diesem Zusammenhang entwickelte Indexstruktur für dynamische hierarchische Daten vor, die allgemeine Baumstrukturen[1] unterstützt und von so genannten Labeling Schemes aus dem XML-Umfeld inspiriert ist. Sie benötigt für Hierarchien mit n Knoten Speicherplatz in der Größenordnung $\mathcal{O}(n)$, unterstützt die genannten Änderungsoperationen in $\mathcal{O}(\log n)$, und erlaubt gleichzeitig diverse Anfragen in $\mathcal{O}(\log n)$ auszuwerten.

2 Kodierung von Hierarchien durch geschachtelte Intervalle

Unser Ansatz ist inspiriert von der bekannten Kodierung einer Baumstruktur durch *geschachtelte Intervalle*. Die Baumkanten werden dabei nicht explizit abgespeichert; stattdessen wird jedem Knoten v ein Intervall $[v.l, v.u]$ zugeordnet, wobei $v.l$ und $v.u$ Ganzzahlen sind und jeweils $[v.l, v.u]$ die Intervalle der direkten und indirekten Kindknoten von v einschließt. (Wir benutzen die Bezeichnungen l und u für „lower“ und „upper“). Im relationalen Schema würden wir die Intervalle in einer Tabelle mit drei Spalten node-id, l und u ablegen. Die Kantenstruktur ist durch die Knotenmarken vollständig kodiert, weshalb dieses und verwandte Verfahren auch unter dem Begriff *Labeling Schemes* bekannt sind. Abbildung 1 zeigt eine Beispielkodierung.

[1] Genauer: *rooted*, *ordered*, *labeled* forests, d. h. Wälder von Bäumen mit jeweils eindeutigem Wurzelknoten und eindeutig festgelegter Reihenfolge von Geschwisterknoten, wobei zusätzlich die Knoten mit Marken versehen werden können.

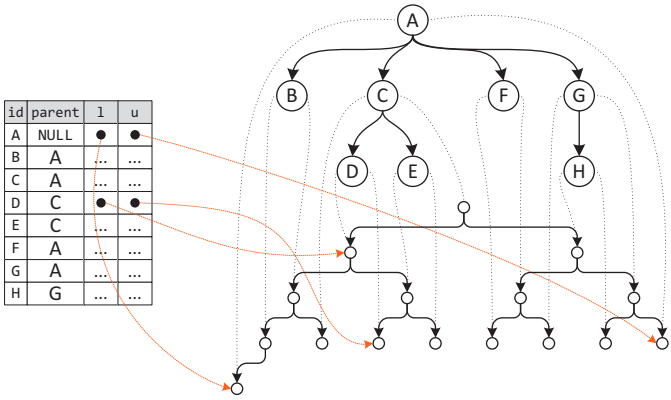

Abbildung 2: Die Beispielhierarchie mit zugehöriger Order-Tree-Repräsentation.

Der größte Vorteil der Intervallkodierung liegt darin, dass man viele Anfragen in $\mathcal{O}(1)$ auswerten kann, indem man die Marken der beteiligten Knoten betrachtet. Speziell für die aus dem XPath-Umfeld bekannten *Axis Checks* entlang verschiedener Achsen der Hierarchie (Vor-/Nachfahren, Cousins etc.) ist dies besonders nützlich, etwa:

$$\text{is-ancestor}(v, w) := v.l < w.l \wedge w.u < v.u$$

Gegenüber Änderungen ist die Kodierung andererseits sehr unflexibel: Möchte man einen neuen Blattknoten an der Stelle l einfügen, muss man alle Intervallgrenzen, die größer oder gleich l sind, um 2 erhöhen, um eine Lücke für das neue Intervall $[l, l + 1]$ zu schaffen. Im Durchschnitt müssen $n/2$ Intervallgrenzen angepasst werden – Einfügen hat somit die Komplexität $\mathcal{O}(n)$. Verschiedene Vorschläge, dieses Problem zu umgehen (vgl. Abschnitt 5), etwa durch Vorreservieren von Lücken oder durch Verwendung von Gleitkommazahlen, können an der Tatsache, dass früher oder später eine Neunummerierung von $\mathcal{O}(n)$ Knoten notwendig wird, nichts ändern.

3 Indexierung hierarchischer Daten durch Order Trees

3.1 Das Konzept des Order Trees

Unsere Idee ist nun, das Problem der Neunummerierungen zu beseitigen, indem wir auf die numerischen Intervallgrenzen ganz verzichten. Dies basiert auf der Beobachtung, dass die absoluten Zahlen für die Beantwortung von Anfragen irrelevant sind; alleine die relative Anordnung der Intervallgrenzen, d. h. die entsprechende Ordnungsrelation $<$, ist entscheidend. Um Änderungen effizient zu unterstützen, kodieren wir die Ordnungsrelation in einer dynamischen, balancierten Baumstruktur, die wir **Order Tree** nennen. Abbildung 2 illustriert deren Aufbau: Jedem Knoten der Hierarchie (oben) werden zwei Intervallgrenzen zugeordnet, denen wiederum zwei Baumknoten im Order Tree (unten) entsprechen. Die Ordnungsrelation auf den Intervallgrenzen ergibt sich aus der Anordnung der Einträge

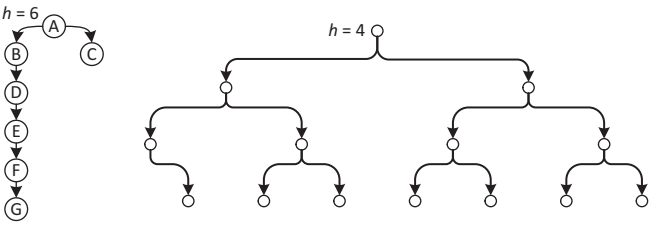

Abbildung 3: Eine degenerierte Hierarchie und die balancierte Struktur des zugehörigen Order Trees.

innerhalb des Order Trees: Eine Intervallgrenze b_1 ist *kleiner* als eine Grenze b_2, wenn sich der entsprechende Order-Tree-Eintrag e_1 *links* vom Eintrag e_2 befindet. Den Datensätzen der zugehörigen Knotentabelle (links) ordnen wir nun, statt numerischer Intervallgrenzen, Zeiger auf die Order-Tree-Einträge zu. Beim ersten Zugriff zur Laufzeit wird der Order Tree aus den Informationen in der Tabelle, insbesondere der parent-Spalte, aufgebaut. Da der Order Tree die hierarchische Struktur vollständig kodiert, können wir fortan alle strukturbezogenen Anfragen und Änderungsoperationen rein auf dem Order Tree durchführen und brauchen die Hierarchie selbst nicht explizit zu materialisieren.

Ein Order Tree R soll folgende Operationen in jeweils $\mathcal{O}(\log n)$ unterstützen:

R.**compare**(e_1, e_2) vergleicht zwei Einträge bezüglich der Ordnungsrelation $<$.

$e' \leftarrow R$.**insert**(before e) fügt einen neuen Eintrag e' unmittelbar vor e hinzu. "before" bezieht sich hierbei auf die Reihenfolge gemäß $<$.

R.**remove**(e) entfernt einen Eintrag e.

R.**move-range**$([e_1, e_2],$ before $e)$ verschiebt alle Einträge im Bereich $[e_1, e_2]$ unmittelbar vor e, vorausgesetzt, dass $e_1 < e_2$ ist und e sich nicht in $[e_1, e_2]$ befindet.

Um das abstrakte Konzept des Order Trees umzusetzen, kann man prinzipiell jede Baumstruktur einsetzen, die folgende Eigenschaften erfüllt: (1.) Im Baum ist die relative Reihenfolge seiner Einträge kodiert; sie bleibt insbesondere unter Balancierungsoperationen erhalten. (2.) Die genannten Operationen lassen sich auf der Baumstruktur mit einer Komplexität von $\mathcal{O}(\log n)$ realisieren. Um das zu garantieren, wird der Baum balanciert gehalten. (3.) Zeiger auf Einträge sind „stabil", werden also nicht durch strukturelle Änderungen invalidiert. — Suchbaumstrukturen wie der AVL- oder der Rot-Schwarz-Baum [CLRS09] bieten sich unmittelbar an, wobei wir auf einen Großteil der üblichen Funktionalität eines Suchbaums verzichten. Insbesondere speichert der Order Tree *keine* Schlüssel in Knoten: alle genannten Operationen beziehen sich auf bereits gegebene Einträge (e, e_1, bzw. e_2); wir müssen also nie nach einem bestimmten Schlüssel suchen.

Eigenschaft 2 garantiert uns ein gutartiges Verhalten auch im schlimmsten Fall – unabhängig davon, wie stark die Struktur der Hierarchie degeneriert sein mag (siehe Abb. 3). Die Komplexität der Algorithmen zum Einfügen und Löschen ist in der Regel $\mathcal{O}(h)$, wobei h die Baumhöhe ist. Da der Order Tree einer Hierarchie mit n Knoten $2n$ Einträge enthält und balanciert ist, ist die Komplexität folglich $\mathcal{O}(\log n)$.

Eigenschaft 3 ist nötig, damit wir in die Hierarchietabelle Zeiger auf die zugehörigen Order-Tree-Einträge verwalten können. Die Implementierung darf also Einträge nicht nach

Belieben im Arbeitsspeicher verschieben. Bei zeigerbasierten Strukturen wie dem AVL- oder Rot-Schwarz-Baum ist das problemlos möglich; bei Array-basierten Strukturen wie dem B-Baum lassen sich stabile Zeiger aber i. d. R. nur erreichen, indem man eine Indirektionsstufe, nämlich eine Zuordnung von (stabilen) Referenzen auf Order-Tree-Einträge zu Speicherbereichen, einführt und bei Änderungsoperationen entsprechend aktualisiert.

3.2 Implementierung der Order-Tree-Operationen

Bei **insert** und **remove** handelt es sich um einfache Varianten der bekannten Suchbaumoperationen, wobei es nicht nötig ist, den referenzierten Eintrag e anhand seines Schlüssels zu suchen, da e als Argument bereits gegeben ist. Die Aufgabe besteht also im Wesentlichen nur noch darin, mittels Rotationen die Balance des Baumes wiederherzustellen.

compare ist eine Operation, die in Suchbäumen üblicherweise nicht unterstützt und auch gar nicht benötigt wird. Wir können sie implementieren, indem wir von den beiden gegebenen Einträgen aus die Elternzeiger verfolgen, bis wir bei der Wurzel ankommen. Anschließend können wir durch Vergleichen der beiden Pfade zur Wurzel feststellen, welcher Eintrag „weiter links“ im Baum liegt. Die Baumknoten müssen dazu Zeiger auf ihre jeweiligen Elternknoten speichern, was sowohl beim AVL- als auch beim Rot-Schwarz-Baum ohnehin üblich ist, nicht jedoch beim B-Baum. Im Falle des B-Baums ist der Zusatzaufwand für die Verwaltung der Elternzeiger aber vernachlässigbar.

Wenn wir zusätzlich in jedem Order-Tree-Knoten die Teilbaumhöhe – d. h., den Abstand zum tiefsten Kind im Teilbaum – speichern, können wir compare dadurch beschleunigen, dass wir nicht bis zur Wurzel nach oben laufen, sondern nur bis zum tiefsten gemeinsamen Vorfahren (*least common ancestor*, LCA) der beiden gegebenen Einträge: Wir laufen zunächst von demjenigen Eintrag aus nach oben, der die geringere Teilbaumhöhe hat, da dieser offenbar weiter vom LCA entfernt ist. Sobald die Teilbaumhöhe der Einträge übereinstimmt, laufen wir von beiden aus simultan nach oben, bis wir beim LCA ankommen. Nun lässt sich leicht feststellen, welcher der beiden Pfade „weiter links“ liegt. Zur Speicherung von Teilbaumhöhen bis zu 256 genügt ein Byte. Sowohl gegenüber den dann überflüssigen Balance-Bits beim AVL- oder Rot-Schwarz-Baum als auch gegenüber der Größe eines B-Baum-Knotens ist dieser zusätzliche Speicheraufwand vernachlässigbar.

move-range implementieren wir dadurch, dass wir sie auf die beiden Operationen split und join auf balancierten Bäumen zurückführen:

$(T_1, T_2) \leftarrow T.$**split**(before e) teilt den Baum T links vom Eintrag e in zwei Teilbäume auf: T_1 enthält die Einträge, die kleiner sind als e, T_2 alle anderen (einschließlich e). Die relative Anordnung der Einträge bleibt dabei erhalten. Außerdem sind sowohl T_1 als auch T_2 balanciert.

$T \leftarrow$ **join**(T_1, T_2) konkateniert T_1 und T_2 unter Beibehaltung der relativen Anordnung der Einträge und so, dass der resultierende Baum T balanciert ist.

Sowohl split als auch join sind recht unbekannte Operationen, es gibt aber Algorithmen in $\mathcal{O}(\log n)$ für die meisten verbreiteten Suchbäume. Mit ihrer Hilfe können wir move-

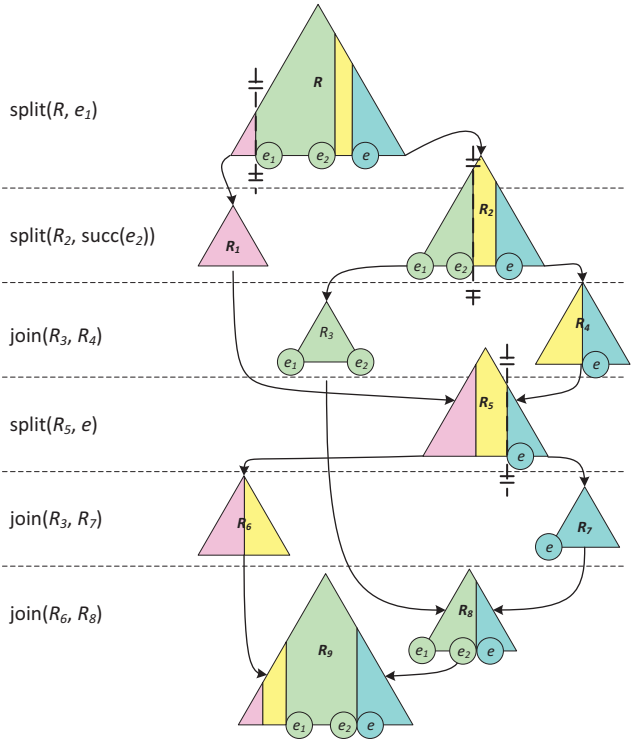

Abbildung 4: Ausführung von move-range$([e_1, e_2]$, before $e)$ auf einem Order Tree.

range in logarithmischer Zeit ausführen (vgl. Abbildung 4): Dazu schneiden wir den zu verschiebenden Bereich $[e_1, e_2]$ mittels split aus, trennen dann den Baum bei e mittels split auf, und setzen schließlich mittels join die entstandenen Teilbäume wieder entsprechend zusammen.

Bulkloading. Mittels insert lässt sich ein Order Tree inkrementell aufbauen, was für n Einträge insgesamt eine Komplexität von $\mathcal{O}(n \log n)$ hat. Wenn aber sämtliche Strukturinformationen der Hierarchie bereits vorliegen, können wir den Baum auch durch „Bulkloading" aufbauen, was mit Hilfe einer Hashtabelle in $\mathcal{O}(n)$ gelingt. Dazu benötigen wir zuerst eine Zwischenrepräsentation der Hierarchie, die uns eine effiziente Tiefensuche erlaubt. Diese erhalten wir in einem ersten Schritt, indem wir eine Hashtabelle von den Knotenschlüsseln zu jeweils einer Kindknotenliste erzeugen, und diese in einem Durchlauf durch die ursprüngliche Hierarchietabelle auffüllen. Einen während dem Durchlauf neu angetroffenen Knoten v der Hierarchietabelle müssen wir jeweils in die Kindliste von dessen Elternknoten p einfügen. Die Hashtabelle liefert uns die Kindliste von p in $\mathcal{O}(1)$. In einem zweiten Schritt führen wir dann anhand der Hashtabelle eine Tiefensuche durch die Hierarchie durch, und erzeugen bei jedem Knoten v *vor* dem Betrachten seiner Kinder einen „unteren" Order-Tree-Eintrag $v.l$ sowie *nach* dem Betrachten seiner Kinder einen „oberen" Eintrag $v.u$. Da dabei die Intervallgrenzen bzw. Order-Tree-Einträge in geordneter Reihenfolge von unten nach oben erzeugt werden, können wir allgemein bekannte Algorithmen

```
function H.insert-node(below v)
    e_1 ← R(H).insert(before v.u)
    e_2 ← R(H).insert(before v.u)
    return (e_1, e_2)

function H.remove-node(v)
    R(H).remove(v.l)
    R(H).remove(v.u)

function H.relocate-subtree(v_1, below v_2)
    R(H).move-range(
            [v_1.l, v_1.u], before v_2.u)

function H.is-ancestor(v_1, v_2)
    return R(H).compare(v_1.l, v_2.l)
        ∧ R(H).compare(v_2.u, v_1.u)

function H.next-sibling(v)
    e ← R(H).succ(v.u)
    if R(H).is-upper-bound(e)
        return ⊥    // no sibling
    return R(H).node(e)
```

Abbildung 5: Umsetzung von Operationen auf einer Hierarchie H in Operationen auf dem zugehörigen Order Tree $R(H)$.

zum Aufbauen eines AVL-, Rot-Schwarz- oder B-Baums in $\mathcal{O}(n)$ einsetzen.

3.3 Der Order Tree als Hierarchie-Index

In Abbildung 5 zeigen wir nun, wie sich Anfragen und Operationen auf einer Hierarchie unmittelbar in Operationen auf dem zugehörigen Order Tree übersetzen lassen. Mit **insert-node** bezeichnen wir das Einfügen eines neuen Blattknotens als letztes Kind unterhalb (below) eines gegebenen Referenzknotens v. Diese Operation entspricht dem Einfügen zweier Einträge in den Order Tree, nämlich der unteren und der oberen Intervallgrenze; sie werden bezüglich der Ordnungsrelation $<$ zu direkten Vorgängern (daher "before") der oberen Intervallgrenze $v.u$ ihres Elternknotens. Einen Knoten zu löschen (**remove-node**) bedeutet, seine zwei Intervallgrenzen $v.l$ und $v.u$ zu löschen. Das Umhängen (**relocate-subtree**) eines Teilbaums mit Wurzel v_1 als letztes Kind unter (below) einen Knoten v_2 übersetzen wir direkt in move-range des Intervalls von v_1 unmittelbar vor die obere Intervallgrenze von v_2. Was strukturbezogene Anfragen betrifft, zeigen wir beispielhaft die Implementierung des Axis-Checks **is-ancestor**. Analog zur Definition in Abschnitt 2 vergleichen wir die Intervallgrenzen der Knoten, wobei wir nun die Vergleichsfunktionalität des Order Trees benutzen. Axis-Checks entlang anderer Achsen lassen sich analog implementieren. Als weiteres Beispiel betrachten wir die Anfrage **next-sibling**, die den nachfolgenden Geschwisterknoten eines Knotens ermittelt. Ihre Implementierung bedient sich der Möglichkeit, im Order Tree in amortisiert konstanter Zeit zum nachfolgenden Eintrag zu gehen (succ) und dann den Zeiger zum zugehörigen Knoten zurückzuverfolgen (node).

4 Auswertung

Um die Anwendbarkeit unseres Ansatzes in der Praxis zu beurteilen, haben wir verschiedene Varianten des Order Trees in C++ implementiert: einerseits basierend auf dem AVL-Baum, andererseits basierend auf dem B-Baum mit wählbarem minimalem Verzweigungs-

grad d, wobei wir für Messungen die Werte $d = 2$, 16 und 128 benutzt haben. Beide Varianten speichern die Teilbaumhöhe h in jedem Knoten. Die B-Baum-Implementierung benutzt zudem eine indirekte Eintragstabelle, um stabile Zeiger auf Einträge zu ermöglichen, wie in Abschnitt 3.1 skizziert. Unsere Messungen haben wir auf einer HP Z600 Workstation durchgeführt, bestückt mit einer 6-kernigen Intel Xeon X5650 CPU (2,66 GHz, 12 MB Cache), 24 GB RAM und SuSE Linux Enterprise Server als Betriebssystem.

Für die Messungen wird eine Hierarchie durch Einfügen an zufälligen Positionen aufgebaut, wobei wir die Knotenanzahl n variieren über $n = 10^4$, 10^6 und 10^8. Nach dem Aufbauen werden mit gleichen Wahrscheinlichkeiten unter Zeitmessung die Order-Tree-Operationen insert, remove oder compare auf jeweils zufällig gewählten Hierarchieknoten ausgeführt. Aus Platzgründen können wir die Ergebnisse an dieser Stelle nicht vollständig wiedergeben, fassen unsere Beobachtungen aber wie folgt zusammen: **(1.)** Bei $n = 10^8$ – einer Größe, welche die wenigsten Anwendungsfälle benötigen werden – erreichen wir absolute Raten von durchschnittlich 1–5 Millionen Operationen pro Sekunde für jede der verglichenen Operationen. Die verschiedenen Order-Tree-Varianten liegen in derselben Größenordnung; insbesondere sind die Zahlen für den AVL-basierten Baum nahezu gleich zu denen des B-Baums mit $d = 2$ (der einem 2-3-4-Baum entspricht). **(2.)** Außerdem liegen die Raten in derselben Größenordnung wie vergleichbare Operationen, die wir auf einem *Cache-sensitiven B$^+$-Baum* [RR00] durchgeführt haben, einer hochperformanten Indexstruktur, die für moderne Hardware optimiert wurde. **(3.)** Bei der B-Baum-Variante lässt sich durch Erhöhen von d die Performanz der compare-Funktion signifikant steigern. Das rührt daher, dass mit größerem d die B-Baum-Höhe geringer wird. Gleichzeitig erhöht sich aber der Verwaltungsaufwand für die Knoten, was Änderungsoperationen merklich verlangsamt. Man kann also durch Anpassen von d einen dem jeweiligen Anwendungsfall angepassten Kompromiss zwischen Änderungs- und Anfrageperformanz erreichen. **(4.)** Durch Erhöhen von n fällt die Performanz etwas schneller ab, als die asymptotische Komplexität von $\mathcal{O}(\log n)$ vermuten lässt. Wir führen dies auf Caching-Effekte zurück: Je mehr Arbeitsspeicher benutzt wird, desto mehr Cache-Misses verursachen die zufällig gewählten Operationen, da immer geringere Anteile der Hierarchie in den Cache passen.

5 Verwandte Arbeiten

Allgemein lässt sich sagen, dass Hierarchien im Kontext von relationalen Datenbanken in der Vergangenheit erstaunlich wenig Aufmerksamkeit durch die Datenbankforschung erfahren haben. Anders ist die Situation bei XML-Datenbanken: Für XML-Dokumente – die inhärent hierarchisch aufgebaut sind – gibt es viele Indexstrukturen, oft aufbauend auf dem relationalen Schema. Eines der am weitesten entwickelten Schemata, das XML-Dokumente auf eine Weise kodiert, dass sämtliche XPath-Achsen unterstützt werden, ist das *Pre/Post*-Labeling-Schema von Grust et al. [GvKT04]. Wie viele vergleichbare Schemata für XML ist es aber nicht dynamisch. In einem späteren Beitrag erreichen Boncz et al. [BMR05] mit MonetDB größere Änderungsraten mit einem modifizierten Pre/Post-Schema.

Weitere bekannte Labeling Schemes sind unter den Begriffen *Prefix-based Encoding* oder *Dewey Encoding* bekannt – siehe die Arbeit zu ORDPATH von O'Neil et al. [OOP$^+$04],

auf der Haustein et al. [HHMW05] weiter aufbauen. Mit dem Vorreservieren von Lücken in Labeling Schemes und speziell der geschachtelten Intervallkodierung befassen sich Yu et al. [YLML05] und Li et al. [LM01]. Trophasko [Tro05] diskutiert numerische Lösungsansätze zum Problem der Neunummerierung, wie *Farey Fractions* oder *Dyadic Fractions*. All diese Ansätze lassen sich jedoch durch ungünstige Einfügesequenzen der Länge n außer Gefecht setzen, so dass eben doch eine Neunummerierung in $\mathcal{O}(n)$ notwendig wird, oder aber die Labels auf Größen proportional zu n anwachsen.

Ein weiterer, in unserem Zusammenhang relevanter Forschungszweig sind *Succinct Data Structures* für Bäume – erinnern doch geschachtelte Intervalle ohne die numerischen Intervallgrenzen stark an eine Baumrepräsentation, die in diesem Bereich unter dem Begriff *Balanced Parentheses* bekannt ist. Statische Kodierungen wurden hier sehr gründlich untersucht (z. B. [GRR06]), sind aber in unserem Kontext irrelevant. In [FM09] untersuchen Farzan et al. auch effizient änderbare Kodierungen. Vielversprechend ist der Ansatz von Navarro et al. [Nav09], eine vergleichsweise elegante dynamische Repräsentation durch einen so genannten *Range-Min-Max-Tree*, welche eine große Menge an Operationen in $\mathcal{O}(\log n)$ unterstützt. Es ist allerdings unklar, inwieweit sich die eher Algorithmentheoretischen Beiträge mit ihren oft recht aufwendigen Konstruktionen in der Praxis implementieren, geschweige denn in Code von akzeptabler Performanz überführen lassen.

6 Zusammenfassung

Der sicherlich interessanteste Beitrag unserer Arbeit besteht darin, dass wir mit dem Order Tree einen generischen Mechanismus entwickelt haben, um beliebig geformte Hierarchien durch einen jederzeit balancierten Baum darzustellen. Dieser Mechanismus erlaubt es uns, gewisse Anfragen auf hierarchischen Daten zu beschleunigen. Soweit wir unsere Indexstruktur in dieser Arbeit konkret vorgestellt haben, unterstützt diese das Einfügen und Löschen von Knoten und das Umhängen von Teilbäumen in jeweils $\mathcal{O}(\log n)$, sowie die üblichen Axis-Check-Anfragen in $\mathcal{O}(\log n)$.

Das Konzept des Order Trees lässt sich unter Wiederverwendung bekannter Baumindexstrukturen umsetzen. Wir haben mit Implementierungen experimentiert, die auf dem AVL-Baum und auf dem B-Baum aufbauen; prinzipiell sind aber auch andere Arten von Baumstrukturen als Grundlage denkbar. Speziell im Datenbankbereich ist es von Vorteil, auf vorhandene Index-Implementierungen zurückgreifen zu können, die gründlich erprobt, optimiert und ausgereift sind.

In Zukunft konzentrieren wir uns zum einen darauf, den Speicherplatzbedarf und die Cache-Effizienz unseres Indexes weiter zu verbessern, indem wir das Datenlayout kompaktifizieren, Zeiger einsparen und Techniken aus dem Gebiet der Cache-conscious Data Structures einsetzen. Zum anderen möchten wir zusätzliche Anfragen auf dem Order Tree implementieren und untersuchen, wo die Grenzen bezüglich der durch den Order Tree theoretisch zu unterstützenden Anfragen liegen.

Literatur

[BMR05] Peter Boncz, Stefan Manegold und Jan Rittinger. Updating the pre/post plane in MonetDB/XQuery. In *Inf. Proc. Int. Workshop XQuery Implementation, Experience, and Perspectives (XIME-P)*, 2005.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 3. Auflage, 2009.

[FM09] Arash Farzan und J. Ian Munro. Dynamic succinct ordered trees. In *Automata, Languages and Programming*, Jgg. 5555 of *Lecture Notes in Computer Science*, Seiten 439–450. Springer, Berlin/Heidelberg, 2009.

[GRR06] Richard F. Geary, Rajeev Raman und Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms (TALG)*, 2(4):510–534, 2006.

[GvKT04] Torsten Grust, Maurice van Keulen und Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Systems (TODS)*, 29:91–131, 2004.

[Hau11] Birgitta Hauser. Hierarchical Queries with DB2 Connect By. `http://www.ibm.com/developerworks/ibmi/library/i-db2connectby/`, 2011.

[HHMW05] Michael P. Haustein, Theo Härder, Christian Mathis und Markus Wagner. DeweyIDs—the key to fine-grained management of XML documents. *Proc. Brasilian Symp. Databases*, 20:85–99, 2005.

[LM01] Quanzhong Li und Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proc. Int. Conf. Very Large Databases (VLDB)*, Seiten 361–370, 2001.

[Nav09] Gonzalo Navarro. Fully-functional static and dynamic succinct trees. *Tech. Rep., abs/0905.0768*, 2009.

[OOP+04] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller und Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *Proc. Int. Conf. Management of Data (SIGMOD)*, Seiten 903–908, 2004.

[Ora10] Oracle Corp. Hierarchical Queries — Oracle Database SQL Language Reference 11g Release 1 (11.1). `http://docs.oracle.com/cd/B28359_01/server.111/b28286/queries003.htm`, 2010.

[RR00] Jun Rao und Kenneth A. Ross. Making B+-trees cache conscious in main memory. In *Proc. Int. Conf. Management of Data (SIGMOD)*, Seiten 475–486. ACM, 2000.

[Tro05] Vadim Tropashko. Nested intervals tree encoding in SQL. *ACM Rec. Int. Conf. Management of Data (SIGMOD)*, 34(2):47–52, 2005.

[YLML05] Jeffrey Xu Yu, Daofeng Luo, Xiaofeng Meng und Hongjun Lu. Dynamically updating XML data: numbering scheme revisited. *World Wide Web: Internet and Web Information Systems*, 8:5–26, 2005.