Sarah Schmidt
Benedikt Fries
Alexander Walz

Universität des Saarlandes 2003

Index Structures for XML Documents

Dozent
Ralf Schenkel

Betreuer
Hanglin Pan

Inhaltsverzeichnis:

1. Abstrakt

Die Anforderungen an Datenbanksysteme halb strukturierte Daten zu verwalten, wachsen. Beispiele sind business-to-business Produktkataloge in denen Datensätze unterschiedlicher Schemata gespeichert und durchsucht werden müssen.
Typische Probleme dabei die Daten in traditionelle Datenbanken zu speichern sind, dass die Daten in eine Menge von Tupeln konvertiert werden müssen und in Tabellen abgespeichert werden. Diese Umwandlung ist nicht trivial und setzt a priori Wissen über das Datenschema voraus und schränkt die Flexibilität bei Datenänderungen ein.

Gleichzeitig wird das Speichern und Durchsuchen von Daten in XML mit der Verbreitung von XML im Internet immer wichtiger. Es gibt mehrere XML Anfragesprachen und ein gemeinsames Merkmal all dieser Sprachen ist, dass sie reguläre Ausdrücke benutzen. Dies ist eine neue Herausforderung was das Speichern und Durchsuchen in XML-Daten betrifft, weil der traditionelle Ansatz mit baumbasierten Datenstrukturen nicht den gewachsenen Anforderungen gerecht wird. Wir diskutieren hier einige neue Verfahren zur Speicherung und Indizierung von XML-Daten.

2. Einführung

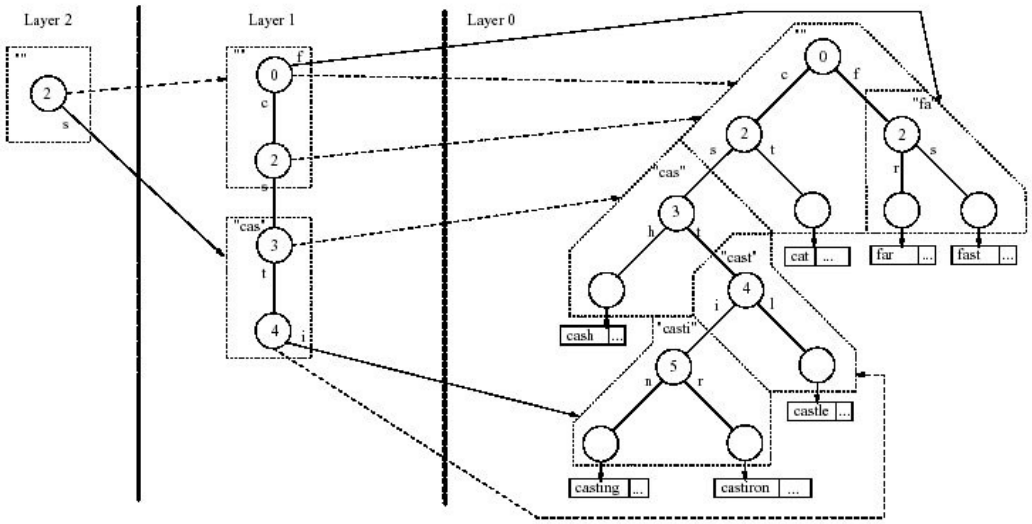
XML ist zum Standard zur Präsentierung und Austausch von Informationen geworden. XML erlaubt den Benutzern ihre eigenen Tags zu definieren. Die Beziehungen der Tags kann durch integrierten Strukturen wie DTD (Document Type Definition) und Referenzen definiert werden. Das Durchsuchen der XML-Daten ist mit verschiedenen Anfragesprachen möglich, wie z.B. XQuery oder XPath.
Trotz der Forschung der letzten Jahre nimmt man an, dass der aktuelle Stand relationaler Datenbanken ungeeignet ist um XML und halbstrukturierte Daten zu speichern. Vor allem, wenn es darum geht reguläre Ausdrücke auszuführen, sind nur wenige konventionelle baumstrukturierte Ansätze bekannt. Diese können aber wegen dem Overhead beim Durchsuchen der Baumstruktur ineffizient sein.
Hier werden einige neue Ansätze zur Speicherung von XML-Daten und halbstrukturierte Daten besprochen, die auf Baumstrukturen und neuen Algorithmen beruhen.

3. Index Fabric

Index Fabric basiert auf Particia Tries. Ein Trie (abgeleitet von Retrival) ist ein Positionsbaum über einem Alphabet A, d. h. dass die Schlüssel nur in den Blättern gespeichert werden.
In einem Particia Trie sind die Knoten mit ihrer Tiefe markiert und die Buchstabenposition wird durch den Knoten repräsentiert.
Particia tries sind unbalanciert und haben ihren Nachteil bei Daten die auf der Festplatte gespeichert werden. Index Fabric hingegen sind balanciert und wie Suchbäume optimiert für die Speicherung auf Festplatten.
Eine Neuerung ist der schichtenbasierte Ansatz bei Index Fabric.
Der String Index wird in Untertries aufgeteilt, die in Blockgröße sind. Diese Blöcke werden indiziert mit einem eigenen Trie auf einem eigenen Block. Diesen Block kann man durch eine neue horizontale Schicht repräsentieren. Die Blöcke werden aufgeteilt, wenn sie zu groß werden und wiederum indiziert mit Hilfe einer zusätzlichen Schicht.

Es gibt zwei Verbindungsarten zwischen den Schichten i und i-1:

- Etikettierte far Verbindungen
- Unetikettierte direkte Verbindungen



Far Verbindungen sind normale Kanten in einem Trie, sie verbinden also Knoten in einer Schicht mit einem Untertrie in einer anderen.
Direkte Verknüpfungen verbinden einen Knoten in einer Schicht mit einem Block mit einem Knoten, der das gleiche Präfix in der nächsten Schicht präsentiert.
Beim Suchen beginnt man beim Wurzelknoten in der Schicht ganz links. Beim Suchen vergleicht man das Suchwort mit den Kantenmarkierungen. Bei einer far Verknüpfung geht die Suche in der nächsten Schicht rechts in einem anderen Block weiter. Passt keine Kantenmarkierung auf das Suchwort so geht die Suche mit einer direkt Verknüpfung weiter und wird dann in der neuen Schicht weitergeführt. Ist die

Suche in der Schicht 0 angelangt, terminiert die Suche entweder mit dem gefundenen Knoten oder der Knoten existiert nicht.
Bei Zugriff auf die Schicht wird jeweils ein I/O Zugriff benötigt.
Ein Vorteil der Particia ist, dass die Schlüssel sehr komprimiert gespeichert werden und dass man viele Schlüssel in einem Block speichern kann. Die Blöcke haben einen hohen Ausgrad (Anzahl der far und direct Verknüpfungen). Die Schicht die am weitesten rechts liegt braucht am meisten Festplattenspeicher, die weiter links liegenden Schichten sind bedeutend kleiner.
Praktisch braucht man nur wenige Schichten um eine hohe Anzahl an Knoten zu speichern, z.B. 3 Schichten um 1 Milliarden Schlüssel zu speichern, dabei könne die Schichten 1 und 2 im Hauptspeicher bleiben und nur die 0-te Schicht muss auf der Festplatte gespeichert werden, was bedeutet, dass man meistens nur einen einzigen Festplattenzugriff braucht, egal wie Lange der Pfad ist.
Alle anderen Operationen wie Suchen, Einfügen, Änderungen und Löschen können ebenfalls effizient implementiert werden. Die Operationen betreffen meistens nur eine Schicht, außer ein Block muss aufgeteilt werden, solche Aufteilungen sind aber selten.

3.1 Indexing XML mit Index Fabric

Damit man XML Daten mit Index Fabric speichern kann führt man Bezeichner für jedes Tag ein.

Bsp:
<invoice>
    <buyer><name>ABC Corp</name></buyer>
<invoice>
wird codiert als **IBN** ABC Corp. Für Attribute wird ein eigener Bezeichner eingeführt, beispielsweise B´ wenn das Tag B ein Attribut hat.

Es gibt zwei Arten von Pfaden:
- raw paths (Wurzel zu Blatt Pfad)
- refined paths (Pfade für häufig vorkommende Anfragemuster)

Index Fabric kann sogar mehrere Dokumente mit unterschiedlicher Struktur indizieren, indem man einfach neue Bezeichner für noch nicht vorhandene Tags einführt.
Refined paths sind Pfade die optimiert für Anfragen mit häufig wiederkehrenden Mustern sind. Sie können auch Suchen mit Wildcards unterstützen. Sie werden vom Datenbankadministrator festgelegt.

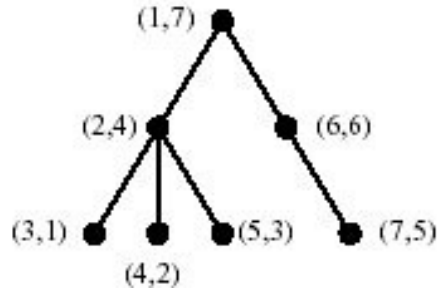
3.2 Experimentelle Erfahrungen

Mit einer bekannten relationalen Datenbank wurde der normale B-Tree Index mit der Index Fabric Methode verglichen. Man führte Anfragen auf der bekannte DBLP (einer Datenbank in der die Veröffentlichungen der Informatik gespeichert werden) aus und kam zum Ergebnis, dass Index Fabric eine Größenordung besser ist als der gewöhnliche RDBMS Ansatz.

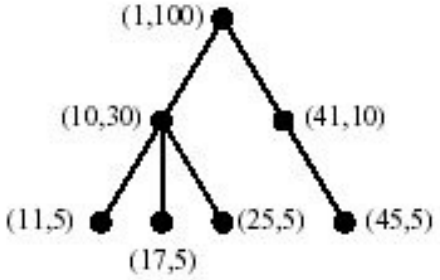
4. XISS Indexing and Storage System

Man kann XML Dokumenten entweder auf eine strukturbasierte, wertbasierte Art oder aber einer Kombination aus beiden durchsuchen. XML Daten werden häufig in einer Baumstruktur organisiert, bei der die Knoten die Tags, Attribute und Text repräsentieren. Um Anfragen schnell durchzuführen muss man die Vorgänger-/Nachfolger Ordnung der Knoten schnell entscheiden können. Hat man dann Teilergebnisse gefunden muss man in der Lage sein diese schnell zusammenzufügen. Das bekannt Nummerierungsschema von Dietz besagt, das ein Knoten x genau dann ein Vorgänger eines Anderen y ist, wenn x in der preorder Reihenfolge vor y und y in der postorder Reihenfolge vor x ist. Dietz Nummerierungsschema kann zwar das Vorgänger/Nachfolgerproblem in konstanter Zeit entscheiden, hat aber den Nachteil, dass es nicht flexibel genug ist. Beim Einfügen eines Knoten muss man durch den ganzen Baum gehen und alle Knoten neu markieren.

Nummerierungsschema nach Dietz



XISS ermöglicht die Suche nach Elementen, Attributen und der Struktur und benutzt ein neues Nummerierungsschema bei dem jeder Knoten mit einem <order, size> Paar markiert wird. Man nummeriert nicht mehr starr durch sondern lässt Raum frei der später genutzt wird.
Nun gilt für jeden Knoten x und seinen Vater y, dass order (x) < order(y) und order(y)+size(y)<=order(x)+size(x).



Für zwei Geschwisterknoten x und y gilt, ist x der Vorgänger von y in der preorder Reihenfolge, dann order(x)+size(x)<order(y).
Daraus folgt für einen Knoten x:
$\text{size}(x) >= \sum ysize(y) \forall y$ die direktes Kind von x sind
Diese Nummerierung ist äquivalent zu der von Dietz. Das Vorgänger/Nachfolgerproblem lässt sich dann wie folgt lösen:
Für 2 Knoten x und y gilt: x ist Vorgänger von y genau dann wenn:
order(x) < order(y)<= order(x) + size(x)

Dieses Schema ist durch die Lücken in der Nummerierung flexibler als das von Dietz und solange noch Lücken da sind kann man Knoten einfügen ohne den ganzen Baum neu durchmarkieren zu müssen.

Die Datenverwaltung in XISS besteht aus 3 Hauptkomponenten.

- element index
- attribute index
- structure index

Daneben gibt es noch den name index und eine Wertetabelle.
Alle unterschiedlichen Namen werden im name index gesammelt, der als B+ Baum implementiert ist. Jeder dieser Namen ist ein name identifier, der von dem name

index vergeben wird. Dies verringert den Overhead bei der Eleminierung der Replikate. Aus dem selben Grund werden alle String-Werte in der Werttabelle gespeichert. Jedes XML-Dokument erhält außerdem eine Dokument ID.
Der Element Index, der Attribute Index und der Struktur Index sind der Mechanismus um die genannte Suchen zu ermöglichen. Der Element Index ermöglicht das schnelle Auffinden von Elementen mit dem gleichen Namen. Jedes Element beinhaltet ein <order,size> Paar. Der Attribut Index funktioniert im Prinzip genauso, er hat nur eine Wert ID. Der Struktur Index funktioniert mit einem Array.

4.1 Pfad Join Algorithmen

Grundsätzlich kann man in Bäumen auf drei Arten suchen, entweder von der Wurzel zum Blatt, vom Blatt zur Wurzel oder man mischt diese Arten und trifft sich in der Mitte. Diese Verfahren garantieren aber nicht, dass effizient gesucht werden kann.

Eine Hauptidee des nun folgenden Verfahrens ist, dass man komplexe Pfad Ausdrücke in mehrere einfachere aufteilen kann. Jeder dieser einfacheren Pfad Ausdrücke führt dann zu einer vorläufigen Lösung die zu der Lösung des komplexeren Ausdrucks zusammengebaut werden kann.
Im Allgemeinen kann man einen regulären Pfadausdruck zu einer Kombination der folgenden Teilausdrücke aufteilen:

1. ein Unterausdruck mit einem einzelnen Element oder Attribute
2. ein Unterausdruck mit einem einzelnen Element und einem Attribute (bsp. f*igure[@caption =“Tree frogs“]*)
3. ein Unterausdruck mit zwei Elementen (bsp. *chapter/figure/* oder *chapter/_*/figure*)
4. ein Unterausdruck der ein Kleene closure (+,*) ist
5. ein Unterausdruck der eine Vereinigung zweier Unterausdrücke ist

Unterausdrücke mit einem Element oder Attribut können direkt durch Zugriff auf den Element oder Attribute Index verarbeitet werden. Eine Vereinigung zweier Unterausdrücke kann durch das Mischen und Gruppieren nach den Dokumenten ausgeführt werden. Für die anderen 3 Arten des Joins gibt es den EA –Join (2), den EE-Join(3) und den KC-Join.

4.1.1 EA-Join Algorithmus

EA-Join Algorithmus

Input: $\{E_1, \ldots, E_m\}$: E_i is a set of elements having a common document identifier;
$\{A_1, \ldots, A_n\}$: A_j is a set of attributes having a common document identifier;
Output: A set of (e, a) pairs such that the element e is the parent of the attribute a.

// Sort-merge $\{E_i\}$ and $\{A_j\}$ by doc. identifier.
1: **foreach** E_i *and* A_j *with the same* `did` **do**
  // Sort-merge E_i and A_j
  // by PARENT-CHILD relationship.
2: **foreach** $e \in E_i$ *and* $a \in A_j$ **do**
3: **if** *(e is a parent of a)* **then** output (e, a);
  end
end

Der EA Join verbindet die vorläufige Ergebnisse zweier Unterausdrücke, die entweder ein Elementenliste und eine Attributliste sind. Beispielsweise ist die Eingabe für den EA Algorithmus beim Ausdruck
figure[@caption="Tree frogs"]
eine Liste von figure Elementen und eine Liste von caption Attributen, die nach Dokumenten geordnet sind.Um die zwei Listen zu verschmelzen werden zunächst die Listen nach der Dokument ID gruppiert verschmolzen.
Danach werden die zusammenpassenden Paare durch Betrachtung der Vorgänger/ Nachfolger Beziehung basierend auf ihrem order Wert verschmolzen.
In Teste war der EA-Join schneller als die top down Methode, hauptsächlich weil die Attribut und element Liste nur einmla duchlaufen werden müssen.

4.1.2 EE-Join Algorithmus

EE-Join Algorithmus

Input: $\{E_1, \ldots, E_m\}$ and $\{F_1, \ldots, F_n\}$: E_i or F_j is a set of elements having a common document identifier.
Output: A set of (e, f) pairs such that the element e is an ancestor of the element f.

// Sort-merge $\{E_i\}$ and $\{F_j\}$ by doc. identifier.
1: **foreach** E_i *and* F_j *with the same* `did` **do**
  // Sort-merge E_i and F_j
  // by ANCESTOR-DESCENDANT relationship.
2: **foreach** $e \in E_i$ *and* $f \in F_j$ **do**
3: **if** *(e is an ancestor of f)* **then** output (e, f);
  end
end

Der EE-Join Algorithmus fügt zwei Ergebnisse zusammen die aus einer Liste von Elementen bestehen. Beispielsweise sucht
chapter/_/figure*
alle chapter figure Paare, die eine Vorgänger-/ Nachfolgerbeziehung haben. Wie der EA- Join Algorithmus kann der EE-Join zwei Mengen auf zwei Ebenen mischen. Erst werden die Mengen nach Dokumenten ID sortiert und dann basierend auf ihrem <order, size> Paar zusammengefügt. Der EE-Join ist in Tests 6 bis 10 mal schneller als die normale bottom up Methode.

4.1.3 KC-Join Algorithmus

KC- Join Algorithmus

Input: $\{E_1, \ldots, E_m\}$, where E_i is a group of elements from an XML document.
Output: A Kleene closure of $\{E_1, \ldots, E_m\}$.

// Apply $\mathcal{EE}$-**Join** algorithm repeatedly.
1: set $i \leftarrow 1$;
2: set $K_i^C \leftarrow \{E_1, \ldots, E_m\}$;
3: **repeat**
4: set $i \leftarrow i + 1$;
5: set $K_i^C \leftarrow \mathcal{EE}$-**Join**$(K_{i-1}^C, K_1^C)$;
until *(K_i^C is empty)*;
6: output union of $K_1^C, K_2^C, \ldots, K_{i-1}^C$;

Der KC Join Algorithmus verbindet Ausdrücke, die entweder keinen einen oder mehrere Unterausdrücke haben. Dabei ruft er in jedem Schritt den EE Join auf, solange bis kein Teilergebnis mehr vorliegt. Beispielsweise enthält das Ergebnis beim Ausdruck *chapter* Chapter/chapter in der nächsten Iteration chapter/chapter/chapter und als Ergebnis die Vereinigung aller vorherigen Ergebnisse.

In Hinblick auf Optimierung ist es sehr wichtig den besten Weg zur Zerlegung in die Unterausdrücke zu finden, um mit Hilfe der genannten Algorithmen schnelles Suchen zu ermöglichen.

5. Covering Indexes for branching path expressions

5.1 Einleitung:
In diesem Paper werden Covering Indexes für Branching Path Expression Queries auf XML Daten vorgestellt. Weiterhin wird untersucht, ob diese Indizes die Auswertung solcher Queries beschleunigen. Die Idee ist, die Knoten der Bäume zu Index-Knoten zusammenzufassen, die Pfade der Bäume jedoch zu erhalten, um dann auf dem wesentlich kleineren Index-Baum die Queries auswerten zu können.

XML Daten werden als gerichteter, knoten-markierter Baum mit einer zusätzlichen Menge von Kanten, sogenannten *idref edges*, dargestellt.

5.2 Branching Path Expressions:
Beispiel:
ROOT/metro/neighborhoods/neighborhood[/business=>hotel]/cultural=>museum
Sucht nach allen Museen, die ein Hotel in der Nachbarschaft haben.

Der *primary path* ist der Teil des Ausdruckes, der übrig bleibt, wenn man den Teil in eckigen Klammern, also die Bedingungen, wegnimmt. In unserem Beispiel ist der primary path also:
ROOT/metro/neighborhoods/neighborhood/cultural=>museum

5.3 Index-Graph:
Ein Index-Graph für einen Daten-Graph G ist der Graph I(G), in welchem wir mit jedem Index-Knoten seinen *extent* assoziieren. Ist A ein Index-Knoten, so ist ext(A), der extent von A, eine Teilmenge der Knoten von G.
Ein Index-Graph *covers* eine Branching Path Expression Query genau dann, wenn das Ergebnis der Auswertung der Query über I(G) dasselbe ist wie das Ergebnis der Auswertung der Query über G.

5.4 Refinement:
Eine Teilmenge P_1 von Daten-Knoten ist ein *refinement* (Verfeinerung) einer weiteren Teilmenge P_2, wenn die folgende Bedingung gilt:
Sind zwei Knoten in P_1 in derselben Äquivalenzklasse, so sind sie es auch in P_2.
Ist P_1 ein refinement von P_2, so ist P_2 coarser (gröber) als P_1.

5.5 1-Index:
Die Idee ist, solche Daten-Knoten zu gruppieren, welche die gleiche Menge an eingehenden Pfaden besitzen. Eine solche ideale Gruppierung zu berechnen ist NP-vollständig. Also verwendet man statt dessen eine Gruppierung namens *Bisimilarity*, welche diese ideale Gruppierung verfeinert.

5.6 Bisimilarity:

Bisimilarity ist eine symmetrische, binäre Relation ≈ über der Menge der Daten-Knoten mit folgenden Eigenschaften:
u ≈ v, gdw.
1. u und v haben dasselbe Label
2. $par_u \approx par_v$, mit par_u = parent von u
3. wenn ein idref edge von u' auf u zeigt, so zeigt auch ein idref edge von v' zu v und u' ≈ v'

5.7 Forward and Backward Index:
Enthält Informationen sowohl über eingehende als auch über ausgehende Pfade. Der F&B-Index wird folgendermaßen berechnet:

1. Als Initialisierung werden die Knoten nach ihrem Label gruppiert.
2. Dann werden alle Kanten in G umgekehrt.
3. Nun wird die Bisimilarity Partition berechnet.
4. Die Kanten werden wieder umgekehrt.
5. Und die Bisimilarity Partition erneut berechnet.
6. Schritte 2-5 werden so lange wiederholt bis sich die aktuelle Teilmenge nicht mehr verändert.

5.8 Stability:
Gegeben sind zwei Mengen von Daten-Knoten A und B.
A ist succ-stable in Bezug auf B gdw. A eine Teilmenge der Nachfolger von B ist oder A und die Menge der Nachfolger von B disjunkt sind.
Analog kann auch der Begriff der pred-stability definiert werden.

Der 1-Index ist der gröbste Index welcher

1. eine Verfeinerung des Label Grouping ist
2. succ-stable ist

Der F&B-Index ist der gröbste Index, der sowohl succ-stable als auch pred-stable ist.

=>	Für einen Daten-Graph G ist der F&B-Index der kleinste Index-Graph, der **alle** branching path expressions covered.

Leider ist der F&B-Index oft sehr groß. Möchte man die Index Größe verringern, muss man in der Menge der Branching Path Expressions, welche der Index abdeckt, Kompromisse schließen, da der F&B-Index der kleinste Index ist, welcher alle Branching Path Expressions abdeckt.

5.9 Index Definition Scheme
Das Index Definition Scheme hat das Ziel, Branching Path Expressions, welche als weniger wichtig betrachtet werden, zu eliminieren, um die Indexgröße zu verringern und so die verbleibenden Branching Path Expressions effizienter bearbeiten zu können.

Hierzu gibt es vier Ansätze:

1. In den Daten gibt es oftmals tags, welche weniger wichtig sind, und so bei der Erstellung des Indexes nicht berücksichtigt werden müssen
2. Branching path expressions geben Tree Edges eine größere Wichtigkeit als Idref Edges. Dieses soll im Index widergespiegelt werden.
3. Die meisten Queries beziehen sich auf kurze Pfade, wohingegen lange Pfade eher selten vorkommen. Deswegen soll lokale Ähnlichkeit ausgenutzt werden.
4. Die Baum-Tiefe zu begrenzen ist ein weiterer Ansatz, um den Index zu verkleinern.

zu 1.
Wir ändern den Baum in der Art, daß solche Knoten, welche tags haben, die weniger wichtig sind, das Label *other* erhalten. Die Knoten, die jetzt das Label *other* haben und nicht auf einem Pfad liegen, welcher zu einem Knoten führt, der indiziert werden soll, werden bei der Erstellung des Indexes ignoriert.

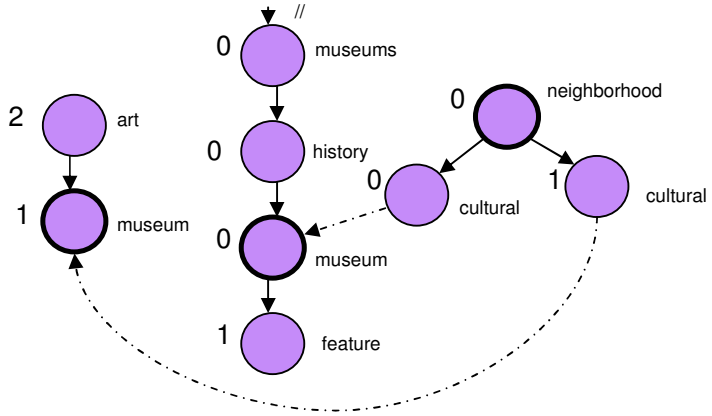
zu 2.
Da den Idref Edges eine geringere Bedeutung zukommen soll als den Tree Edges, spezifizieren wir die Menge der Idref Edges, die wir erhalten möchten. Alle anderen Idref Edges werden bei der Indizierung ignoriert.

zu 3.
Hierzu ist zuerst die Klärung eines weiteren Begriffs der *k-bisimilarity* nötig.

k-bisimilarity $\approx^k$:
1. $u \approx^0 v$ gdw. u und v dasselbe Label haben
2. $u \approx^k v$ gdw.
   * $u \approx^{k-1} v$
   * $par_u \approx^{k-1} par_v$
   * für jedes u' das auf u durch eine Idref Kante zeigt, gibt es ein v', das auf v durch eine Idref Kante zeigt und $u' \approx^{k-1} v'$

Da sich die meisten Queries auf kurze Pfade beziehen, ist es nicht sehr sinnvoll die Index-Teilmengen entlang langer Pfade aufzuteilen.
Daraus ergibt sich, daß es sinnvoll ist, nur Knoten auf Pfaden der Länge bis maximal k zu gruppieren, wobei k beliebig zu wählen ist. Dies kann durch eine Aufteilung der Index-Mengen basierend auf der k-bisimulation erreicht werden. Dieser Index wird A(k)-Index genannt.
So wäre es zum Beispiel sinnvoll, in Rückwärtsrichtung ‚globale' Ähnlichkeit zu fordern, in Vorwärtsrichtung hingegen nur eine Ähnlichkeit auf einer Pfadlänge von höchstens 1.

zu 4.
Baum-Tiefe:



Knoten auf dem Primary Path haben Baumtiefe 0, ebenso Pfade, die eine Kante zu dem primary path hin haben. Knoten auf welche eine Kante von dem primary path zeigt, haben Baumtiefe 1 und so weiter.

Führen wir von dem Algorithmus des F&B-Index nur eine Iteration aus, also Schritt 1-5, so erhalten wir den F+B-Index. Dieser deckt Branching Path Expressions mit einer Baumtiefe von maximal 1 ab. Der F+B+F+B-Index, welchen man nach zwei Iterationen erhält, deckt Queries einer Baumtiefe von maximal 3 ab und so weiter.

Unsere Intuition ist, daß eher selten Branching Path Expression Qqueries mit hoher Baumtiefe auftreten werden.

5.10 Putting it together

Eine Index Definition setzt sich nun aus den folgenden Bestandteilen zusammen:

1. Einer Menge von tags, welche indiziert werden sollen. Nennen wir diese Menge T.
2. Für jeweils die Vorwärts- und Rückwärtsrichtung:
    a. Eine Menge von idref edges, welche bei der Indizierung berücksichtigt werden sollen (ref_{fwd}, ref_{back})
    b. Ein Parameter k, der das Ausmaß der lokalen Ähnlichkeit angibt (k_{fwd}, k_{back})
3. Die Anzahl der Iterationen in der Berechnung des F&B-Index (td = tree depth)

Mit dieser Index Definition können wir nun verschiedene Indizes erzeugen.

5.11 Beispiele:

1. Den F&B-Index erhalten wir, indem wir alle tags und alle idref edges indizieren und
   $k_{fwd} = k_{back} = td = \infty$ setzen
2. Der 1-Index kann erzeugt werden, indem wir alle tags und alle idref edges indizieren und $k_{fwd} = 0$, $k_{back} = \infty$ und $td = 0$ setzen
3. Den A(k)-Index erhalten wir, indem wir alle tags und alle idref edges indizieren und $k_{fwd} = 0$, $k_{back} = k$ und $td = 0$ setzen

Den passenden Index aber nun zu finden, hängt von den zugrundeliegenden Daten und den Queries ab und ist sehr kompliziert. Dies ist Thema zukünftiger Forschung.

6. Updates for Structure Indexes

6.1 Einleitung:

In diesem Paper geht es um Updates für Structure Indexes. Hat man eine Datenbank, auf welcher ein Index schon berechnet wurde und sich nun etwas an diesen Daten ändert, so möchte man nicht den kompletten Index wieder neu berechnen.

Wir betrachten Algorithmen für zwei Arten von Updates:

1. Das Hinzufügen eines Teilbaum, welches dem Hinzufügen einer neuen Datei zu der Datenbank entspricht.
2. Das Hinzufügen einer Kante.

Diese Veränderungen betrachten wir jeweils auf drei verschiedenen Indizes:

1. 1-Index
2. F&B-Index
3. A(k)-Index

Der einzig sonst bekannte Update Algorithmus, im Kontext von Structure Indexes, bezieht sich auf den Strong Data Guide. Dieser basiert aber nicht auf der Bisimulation und kann damit nicht auf die hier betrachteten Indizes übertragen werden.

6.2 1-Index: Hinzufügen eines Teilbaums

G ist der Graph, der den Daten vor dem Hinzufügen eines neues Teilbaums entspricht. I_G ist der darauf berechnete 1-Index. Wir möchten nun einen Teilbaum H unter der Wurzel von G einfügen. Wir berechnen nun den 1-Index auf dem Teilbaum, I_H. Diesen fügen wir jetzt unter der Wurzel von I_G ein. Was wir erhalten, I', ist eine Verfeinerung des tatsächlichen 1-Index, I_{new}. Mit dem folgenden Theorem können wir I_{new} berechnen.

Theorem 1:
Sei G ein Daten Graph, Bisim(G) der darauf berechnete 1-Index und $Bisim^{ref}(G)$ der Index Graph, der auf einer Verfeinerung von G berechnet wurde. Dann gilt:

$$Bisim(Bisim^{ref}(G)) = Bisim(G)$$

Dieses Theorem können wir nun auf I' anwenden und erhalten somit I_{new}, ohne den Index ganz neu zu berechnen.
Die Anzahl der Knoten in I_G, H und I_H seien n_{IG}, n_H und n_{IH}, die Anzahl der Kanten seien m_{IG}, m_H und M_{IH}. Der Algorithmus um einen Teilbaum hinzuzufügen, kann in $O(m_H \log n_H + (m_{IH} + m_{IG}) \log (n_{IH} + n_{IG}))$ ausgeführt werden.

6.3 1-Index: Hinzufügen einer Kante

Eine kleine Änderung wie das Hinzufügen einer Kante kann beliebig große Auswirkung auf den Index-Graph haben.
Sei G der Daten-Graph, welcher durch das Hinzufügen einer Kante von u zu v modifiziert wird. Der veränderte Daten-Graph sei G'. Der 1-Index über G sei Bisim(G) und über G' Bisim(G').
Der hier vorgestellte update Algorithmus, *propagate*, nimmt Bisim(G) als Eingabe und liefert uns $Bisim^{ref}(G')$, eine Verfeinerung des tatsächlichen 1-Index Bisim(G').
Bisim(G') kann dann mit oben vorgestelltem Theorem 1 berechnet werden.

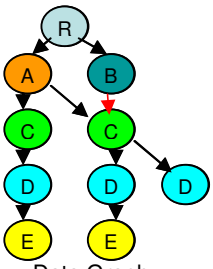
Im ersten Teil dieser Zusammenfassung haben wir den Begriff der Stability definiert.
Ist der Index-Knoten A unstable im Bezug auf Index-Knoten B, so können wir A stabilisieren, indem wir den Index-Knoten in zwei neue Knoten A_1 und A_2 aufsplitten, wobei der

$$ext(A_1) = ext(A) \cap Succ(ext(B)) \text{ und } ext(A_2) = ext(A)\text{-}ext(A_1).$$

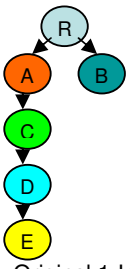
Die originale Bisimulation Partition ist stable. Fügt man eine Kante von u zu v hinzu, so kann es passieren, daß der Index-Knoten mit v in seinem extent, I[v], nicht länger stable ist im Bezug auf I,[u]. Also überprüft *propagate* zuerst, ob es schon eine Kante von I[u] zu I[v] gibt. Falls dies zutrifft, so ist die Stabilitätsbedingung erfüllt und *propagate* terminiert. Ist dies nicht der Fall, entfernt *propagate* v aus ext(I[v]) und erstellt einen neuen Index-Knoten I', welcher nur v in seinem extent enthält.
Jetzt kann es sein, daß die Index-Knoten, welche Kinder von v enthalten, nicht mehr länger stable in Bezug auf alle anderen Knoten sind. Ist das der Fall, so entfernt propagate diese Knoten aus dem extent des Index-Knotens, welcher sie enthält, und fügt sie in neue Index-Knoten ein. Waren zwei Kinder von v zuvor bisimilar, so versucht *propagate,* sie in denselben Index-Knoten einzufügen. Dann geht der Algorithmus weiter zu den ‚Enkeln' von v und so weiter. *Propagate* terminiert, wenn die aktuelle Partition stable ist.

6.4 Beispiel:

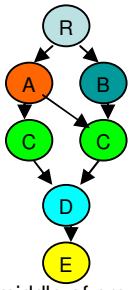
Die rote Kante ist die Kante, welche neu hinzugefügt wird.



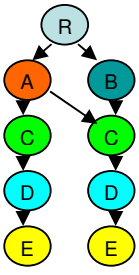
Data Graph



Original 1-Index



In the middle of propagate



Final 1-Index

6.5 Paige and Tarjan:
Der Algorithmus von Paige und Tarjan zur Berechnung der Bisimulation Partition ist sehr effizient. Er funktioniert folgendermaßen, wobei gilt:

X und Q sind Partitionen von Daten-Knoten, mit
1. Q ist eine Verfeinerung von X
2. Q ist stable in Bezug auf X

Ein compound_X_block ist eine X Partition, welche in Q verfeinert ist.

procedure compute_bisim(graph G)
begin
1. Initialize Q by partitioning the data nodes by label
2. Initialize X to a single partition with all data nodes
3. Initialize compound_X_blocks appropriately
4. **while** there is a compound_X_block **do**
5. pick partition x ∈ X which is compound
6. pick partition q ∈ Q contained in X and smaller than half its size
7. replace x in X by q and x-q
8. stabilize Q w.r.t q and x-q

end

Nun können wir auch den *propagate*-Algorithmus mit Hilfe des Algorithmus von Paige und Tarjan genauer angeben:

procedure propagate-brief(u,v)
// Edge added from u to v
begin
1. add an edge from u to v in the data graph
2. **if** there is already an edge between I[u] and I[v] **then**
3. return;
4. Initialize X to be the old partitioning of nodes
5. Create a new index node I'
6. Put v in the extent of I'
7. Add an edge from I[u] to I'
8. Initialize Q to be this partitioning of nodes
9. Run PT (Paige and Tarjan)

end

Theorem 2:
Die Partition, welche *propagate* erstellt, ist stable und außerdem ist sie eine Verfeinerung des tatsächlichen Bisimulation Index.

Nun können wir also mit Theorem 1 den tatsächlichen Bisimulation Index berechnen. Wie aber in Experimenten festgestellt wurde, unterscheidet sich die Verfeinerung, welche uns propagate liefert, nur um etwa 5% von dem tatsächlichen 1-Index. Deswegen können wir in der Implementierung ein ***Lazy Update Sheme*** verwenden, welches den tatsächlichen 1-Index periodisch, also nicht nach jedem Update, berechnet. Damit gehen wir keine Kompromisse in der Genauigkeit des Index ein, da

eine Verfeinerung des tatsächlichen Index genauso präzise ist wie der tatsächliche Index.

6.7 F&B-Index: Hinzufügen eines Teilbaums und Hinzufügen einer Kante

Für den F&B-Index gilt Theorem 1 analog:

Theorem 3:
Sei G ein Daten-Graph, FB(G) der dazugehörige F&B-Index und $FB^{ref}(G)$ der Index-Graph welcher auf einer Verfeinerung von G berechnet wurde. Dann gilt:

$$FB(FB^{ref}(G)) = FB(G)$$

Wenn also ein Teilbaum oder eine Kante hinzugefügt werden soll, ist das Vorgehen analog zu dem Vorgehen bei dem 1-Index.

6.8 A(k)-Index: Hinzufügen eines Teilbaums und Hinzufügen einer Kante

Der Algorithmus, um einen Teilbaum einzufügen, kann analog übernommen werden, da das Theorem 1 auch für den A(k)-Index gilt. Es basiert auf dem Begriff der k-Bisimilarity, welcher im ersten Teil dieser Zusammenfassung definiert wurde.

Theorem 4:
Sei G ein Daten Graph, k-Bisim(G) der darauf berechnete A(k)-Index und $k\text{-Bisim}^{ref}(G)$ der Index Graph, der auf einer Verfeinerung von G berechnet wurde. Dann gilt:

$$k\text{-Bisim}(k\text{-Bisim}^{ref}(G)) = k\text{-Bisim}(G)$$

Für das Hinzufügen einer Kante kann der Algorithmus über dem 1-Index nicht einfach übernommen werden. Anders als bei dem 1-Index oder F&B-Index ist die Auswirkung hier nicht beliebig groß, sondern nur lokal. Hierzu betrachten wir den folgenden Fall:

Seien v,x,y drei Knoten, so daß der kürzeste Pfad von x zu v oder von y zu v mehr als k Kanten enthält. Wird nun eine Kante von einem Knoten u zu v hinzugefügt (oder entfernt), so hat diese Veränderung keinen Einfluß auf die k-bisimulation Beziehung zwischen x und y.

Ein Algorithmus, der sich hier anbietet, wäre die lokale Variante von propagate. Dazu wären aber Informationen über die (k-1)-bisimilarity Beziehung zweier Knoten nötig, um den exakten A(k)-Index zu berechnen. Da dem Algorithmus diese Informationen nicht zur Verfügung stehen, splittet er die Knoten unnötig oft und liefert uns damit eine zu große Anzahl von Index-Knoten.
Dies ist ein Thema für zukünftige Forschung.

6.9 Schlussfolgerung

Das Index Definition Scheme ist sehr flexibel. Sind sowohl die zugrundeliegenden Daten bekannt, als auch auftretende Queries vorhersagbar, so kann das Index Definition Scheme so angepasst werden, daß die Auswertung der Queries sehr beschleunigt werden kann und effizient ist. Sind die Queries jedoch nicht bekannt und muß der Index sehr allgemein gehalten werden, so wird er schnell zu groß, wie zum Beispiel bei dem F&B-Index der Fall. Der große Vorteil dieser Indizes sind jedoch die Update Algorithmen, da es die einzig bekannten Update Algorithmen im Kontext von Structure Indexes sind. Die Update Algorithmen, die hier vorgestellt wurden, arbeiten nachweislich 50 - 80% schneller, als wenn man den Index ganz neu berechnen müsste.

7. Accelerating XPath Location Steps

(Thorsten Grust 2002)

7.1 Einleitung

In seinem Vorschlag für einen für XML optimierten Index konzentriert sich Thorsten Grust auf die XPath-Unterstützung, da bisherige Indizes nicht alle im XPath Standard vorgesehen Achsen unterstützen. Zudem kann dieser Queries durch reine SQL-Anfragen verwirklichen und ermöglicht dadurch einfache und in der Praxis direkt einsetzbare effiziente Umsetzungen, da die ursprünglichen Effizienzoptimierungen der Datenbanksysteme genutzt werden können.

Prinzipiell codiert Grust jeden XML-Tag in einen Knoten, zu welchen er die Preorder- und Postorder-Ränge berechnet. Diese Ränge basieren vereinfacht dargestellt auf einem Durchzählen der Knoten von der Wurzel zu den Blätter, wobei bei der Preorder erst (->pre) der Elternknoten und dann die Kinder von Links nach Rechts durchgezählt werden, während bei Postorder der Elternknoten erst seine Nummer erhält nachdem (->post) bereits dessen Kinderknoten von links nach rechts durchnummeriert wurden.

7.2 Windows

Mittels dieser Zahlen kann man nun die verschiedenen XPath-Achsen darstellen. Dazu wird jeder einzelnen Achse ein Bereich, ein sogenantes Window, zugeordnet der durch Bedingungen bezüglich Pre- und Postorder beschränkt wird. So können zum Beispiel die Nachfolger eines Knotens v bestimmt werden in dem man alle Knoten nimmt deren Preorder Rang größer oder gleich dem von v ist und dessen Postorder Rang kleiner oder gleich dem von v ist. Ebenso kann man die Windows für die anderen X-Path Achsen bestimmen.

7.3 Abspeichern der Daten

Allerdings reichen diese Informationen noch nicht aus um alle Beziehungen zu berechnen. Dazu wird zu jedem Knoten noch der Elternknoten (soweit vorhanden) abgespeichert, was z.B. eine direkte Abfrage der Child-Beziehung ermöglicht. Da im XPath-Standard auch eine Attribute-Achse vorgesehen speichern wir auch ab ob und wenn welchen Tag der Knoten enthält und können somit wirklich jede XPath-Abfrage lösen.
Diese Informationen speichern wir in der „accel“-Tabelle. Da wir hier jeden Tag abspeichern und sogar zu jedem Knoten zusätzliche Informationen wird sie verhältnismäßig groß (linear zur Größe des Ausgangsdokument). Die eigentlichen Daten werden nun in einer weiteren Tabelle abgespeichert, in der der eindeutige Preorder-Rang als Schlüssel dient (ebenso könnte aber auch der Postorder-Rang dienen). Es ist zwar auch möglich die Daten direkt in der „accel“-Tabelle abzuspeichern, aber dies ist unübersichtlicher und auch schwerer zu verwalten.

7.4 Erstellen des Index

Der Algorithmus zum Erstellen der Tabellen unterscheidet zwischen öffnenden und schließenden XML Tags, wobei die „accel“-Tabelle als Stack gespeichert wird. Während eines öffnenden Tags wird der Preorder-Rang, der Preorder Rang des Elternknoten, sowie die Tags berechnet, der fehlende Postorder Rang dann für den schließenden Tag. Da das Dokument dabei nur einmal durchlaufen werden muss ist die Laufzeit linear.

7.5 Anfragen

Queries werden komplett in SQL formuliert. Dazu transformiert man die XPath-Achsen in die entsprechenden Bedingungen für das zugehörige Window. Dabei geht Grust allerdings nicht näher auf die von ihm intuitiv vorgenommenen Joins ein, obwohl hier ähnlich zu dem vorher erwähnten XISS System – je nach Art der Anfrage - auch intuitiv die Query in Teilanfragen aufgespalten wird, die dann wieder vereinigt werden müssen. Dies liegt an der Tatsache, dass die ursprüngliche XML-Struktur in eine von bisherigen Systemen direkt verwaltbare und abfragbare Struktur überführt urde, so dass die Anfrageproblematik mit der bestehender Systeme identisch ist.

7.6 Optimierung

Es gibt zwei generelle Richtungen durch die die Effizienz noch verbessert werden kann. Entweder durch Verbesserung der zugrunde liegenden Strukturen mittels der Verwendung von R- oder B-Tree oder durch weitere Verkleinerung der Windows.

7.7 R-Tree und B-Tree Optimierung

Durch das verwenden von R- oder B-Tree kann die eigentliche Anfrage besonders schnell beantwortet werden. Dafür werden die Knoten temporär in ihrerer Preorder Reihenfolge geordnet und in dieser eingefügt, was zu einer optimalen Speicherplatzausnutzung führt, so dass nur extrem wenige Blätter notwendig sind.
Ein weiterer Vorteil ist, dass augenscheinlich auch die Ergebnisse geordnet sind, was für XQuery notwendig ist. Allerdings ist dieses Verhalten bei R-Trees zwar praktisch beobachtbar, allerdings nicht theoretisch sicher.
Sollten keine R-Trees von der verwendeten Datenbank unterstützt werden, so führt eine Kombination von zwei B-Trees, die einmal in Pre- und einmal in Postorder Richtung erstellt werden für eine vergleichbare Beschleunigung. Zudem ist die Ordnung der Ergebnisse für XQuery diesmal sicher.

7.8 Optimierung durch Einschränken der XPath windows

Die folgende Optimierung schränkt das descendant window weiter ein, was sich auf die descendent-or-self -, die child- und die attribute-Achse positiv auswirkt, da jeweils weniger Knoten bearbeitet werden müssen.
Um dies durchzuführen muss zunächst die Größe eines Knotens v definiert werden.

Def.: Size(v) ist die Größe des Subbaums dessen Wurzel v ist

In diesem Dokument als gegeben angenommen wird nun folgende Gleichung:

(1) $pre(v) - post(v) + size(v) = level(v)$

Für ein Blatt v' wissen wir nun das die Größe size(v')=0 sein muss, so das gilt

(2) $pre(v') - post(v') = level(v') <= height(t)$

Für das am weitesten rechts gelegene Blatt v gilt nun zusätzlich

(3) $post(v) = post(v') + (level(v') - level(v))$

Mittels (2) und (3) kann nun der Preorderrang besser abgeschätzt werden:

(4) $pre(v') <= post(v) + height(t)$

Ein analoges Vorgehen bezügliches des am weitesten links gelegenen Blattes führt zu einer vergleichbaren Abschätzung bezüglich des Postorderranges

(5) *post(v“) >= pre(v) - height(t)*

Insgesamt kann damit das descendent window sehr stark eingeschränkt werden, was man in einer direkten Gegenüberstellung besonders gut erkennt

Das ursprüngliche:

descendant of v = pre is between (pre(v) , end) AND
post is between (0, post(v))

Nach der Optimierung

descendant of v = pre is between (pre(v) , post(v) + height(t)) AND
post is between (pre(v) - height(t), post(v))

7.9 Fazit

Die größte Stärke des Index besteht in der Unterstützung der kompletten XPath Spezifikation und der weitgehenden XQuery-Unterstützung. Zudem erlauben die Pre- und Postorder Rangangaben eine besonders effiziente Bearbeitung von Erreichbarkeitsanfragen und auch das Erstellen ist mit linearem Aufwand möglich. Der große Schwachpunkt ist die mangelnde Updatemöglichkeit. Grust schweigt dazu, so dass der vorhandene Index komplett neu erstellt werden muss. Zudem ist der Platzbedarf verhältnismäßig groß, da jeder Tag als Knoten abgespeichert wird.

8. APEX: An Adaptive Path Index for XML data

(by Chin-Wan Chung, Jun-Ki Min, Kyuseok Shim, 2002)

8.1 Einleitung

Im Gegensatz zu den letzten Indizes liegt der Schwerpunkt des APEX Index auf der effizienten Verarbeitung von Teilpfaden und deren kompaktes Speichern.
Um diesen Effekt zu erreichen besteht der Index aus zwei Teilen – dem gerichteten Graphen GAPEX und dem Hash Tree HAPEX. Dabei bewahrt GAPEX die generelle Struktur des Dokuments, während HAPEX zum Bearbeiten von Pfadanfragen verwendet wird.

8.2 Die Struktur

Dabei fasst Apex für GAPEX die Baumdarstellung eines XML Dokuments anhand der eingehenden Kanten zusammen, so dass ein Knoten in GAPEX mehrere Knoten in GXML darstellen kann. Aus diesem Grund wird jedem Knoten ein Extent zugeordnet. In HAPEX wird jedem eingehenden Pfad entweder direkt ein Knoten aus GAPEX zugeordnet, ist dies nicht möglich, da 2 Knoten nA, nB in GAPEX gleich benannte eingehende Kanten nA1,nB1 haben, aber nicht identisch sind müssen sie anhand der vorherigen Kanten nA0, nB0 unterschieden werden. Dieser Rückblick erfolgt durch das Einfügen eines neuen Knotens in HAPEX, der die vorherigen Label nA0, nB0 auf die zugehörigen Knoten auf nA, nB abbildet.

8.3 Behandlung von Anfragen

Anhand von HAPEX ist offensichtlich, dass direkte Pfadangaben extrem schnell bearbeitet werden können. Dazu wird einfach der gesuchte Pfad rückwärts in HAPEX gesucht, also zuerst das letzte Label. Dabei muss nur so lange nachgeschlagen werden bis direkt dem gesuchten Pfadteil ein Knoten aus GAPEX zugeordnet ist.

8.4 Der Remainder

Spätestens hier fällt nun eine Besonderheit auf, nämlich der Remainder. Dieser wird bei Pfaden in HAPEX deren Länge größer eins ist verwendet und dient als Platzhalter für alle anderen Möglichkeiten. Dies mag zwar auf den ersten Blick seltsam erscheinen, aber da eines der Hauptziele ja Effizienz ist wird der Index auf optimale Performance für häufig genutzte Pfade optimiert. Der Remainder stellt einen Knoten dar der nicht zu den häufig genutzten Pfaden gehört, deshalb auch die Bezeichnung.

8.5 Konstruktion und Verwaltung

APEX ist darauf ausgelegt für einem bestimmten Workload optimale Ergebnisse zu liefern. Da sich aber die Art der Anfragen in der Lebenszeit eines Indizes ändern kann, ist es möglich APEX an einen veränderten Workload anzupassen. Damit befindet sich APEX in einem dauernden Prozess.
Zunächst wird ein erster unoptimierter Index, genannt APEX-0, erstellt. Im folgenden Schritt werden die häufig genutzten Pfade bestimmt, was natürlich eine repräsentative Nutzung voraussetzt. Hat man die häufig genutzten Pfade, so kann der Index daran angepasst werden. Sollte nun im folgenden Änderungen auftreten wiederholt man einfach die letzten beiden Punkte.

8.6 Erstellen des ersten Index APEX-0

Der Algorithmus zum Erstellen des Ausgangsindex verarbeitet die Baumdarstellung eines XML Dokuments ausgehend von der Wurzel und läuft solange bis in einem Durchlauf keine neuen Kanten und Knoten hinzugefügt wurden.
Da in jedem Durchlauf genau ein Knoten betrachtet wird, werden zunächst alle ausgehenden Kanten nach ihrer Beschriftung sortiert. Für jedes dieser Label wird nun eine Kante in GAPEX und ein Eintrag in HAPEX erstellt, wobei der Anfang der Kante natürlich der aktuelle Knoten in GAPEX ist. Sollte nun der Endknoten noch nicht in GAPEX existieren wird er neu erstellt. Schlußendlich wird der Algorithmus mit den neu hinzugefügten Knoten und den neu hinzugefügten Kanten rekursiv neu gestartet.
Zu beachten ist hierbei, dass die idref Beziehungen des XML-Dokumentes durch Kanten im XML Baum dargestellt werden, deren Label mit einem @ beginnt.

8.8 Identifizierung von häufig genutzten Pfaden

Um häufig genutzte Pfade zu identifizieren, muss während der Anfragen mitgezählt werden, wie häufig jedes Label in GAPEX aufgerufen worden ist. Hierbei gibt es das Problem, dass Teilpfade leicht eine zu hohe Wertung erhalten können. Wenn zum Beispiel der einzige häufig genutzte Pfad A-B-C wäre, so muss vermieden werden, dass auch die Teilpfade A-B und B-C als häufig angesehen werden. Ebenso problematisch ist es, die Grenze minSUP festzulegen, ab der ein Pfad als häufig eingestuft wird. Gerade durch die Anpassung von minSUP an die Anforderungen kann die Effizienz enorm gesteigert werden.
Folglich wird HAPEX um ein zusätzliches Zählfeld erweitert, in welchem die Zugriffe passend aufaddiert werden. Pfade der Länge eins werden unangetastet gelassen, unabhängig von ihrer Häufigkeit, da sonst die Fähigkeiten des Index eingeschränkt werden. Nun werden alle Pfade mit einer geringeren Häufigkeit als minSUP gelöscht. Sollte nun in der Erweiterung eines Knotens von HAPEX etwas geändert und deshalb auch der Remainder geändert werden muss, wird dieser zunächst auf NULL gesetzt, da die dafür nötige Anpassung im nächsten Schritt geschieht.

8.9 Update mit häufig genutzten Pfaden

Wenn die häufig genutzten Pfade identifiziert und der Baum daraufhin bereinigt wurde, muss er nun aktualisiert werden. Dazu werden alle Knoten von der Wurzel ausgehend durchlaufen und damit klar ist, ob ein Knoten schon bearbeitet wurde oder nicht erhalten sie ein boolsches „visited“ Feld, welches zu Beginn mit false initialisiert wird.
Zu Beginn des Algorithmus ist fraglich, ob im letzten Schritt neue Kanten hinzugefügt wurden. Ist dies nicht der Fall und der betrachtete Knoten ist schon bearbeitet worden, so endet das Update.
Sollten Kanten hinzugefügt worden sein, so werden sie analog zum Erstellungs-algorithmus zunächst nach Labeln sortiert und für jeden Zielknoten der Kanten der Algorithmus fortgesetzt, wie wenn weder Kanten hinzugefügt noch der aktuelle Knoten bisher bearbeit wurde.
Es werden die ausgehenden Kanten bestimmt und für jedes Label wird der komplette Pfad bestimmt. Sollte es diesen Pfad noch nicht in HAPEX geben, so wird ein neuer Knoten erstellt. Gibt es ihn schon und sollte der Zielpunkt der Kante nicht mit dem bereits vorhanden Knoten übereinstimmen, so werden die Kanten berechnet, die dem Knoten hinzugefügt werden müssen.

Danach wird eine neue Kante zwischen dem aktuellen und dem Zielknoten erstellt und GAPEX hinzugefügt. Sollte es bereits eine Kante mit diesem Label geben, so wird diese entfernt.
Nachdem der Pfad in Hapex eingetragen wurde wird der Algorithmus rekursiv mit dem Endknoten, den hinzugefügten Kanten und dem neuen Gesamtpfad aufgerufen.

8.10 Fazit

APEX ist ein effizienter Index wenn der Schwerpunkt der Anfragen an die Datenbank aus Pfadanfragen besteht und wenn konstante häufig genutzte Pfade vorliegen. Trotz seiner Anpassungsfähigkeit ist er ineffizient für einen sich schnell ändernden Workload. Zudem ist die Optimierung schwierig, da es nicht klar ist, wann genau ein Update notwendig ist und wie am besten minSUP festgelegt werden sollte. Zudem ist das Erstellen und Updaten sehr aufwendig und es ist nicht möglich, ohne ein erneutes Aktualisieren des gesamten Index (und damit eigentlich einem kompletten Neuerstellen) Dokumente hinzuzufügen oder Pfade zu ändern.

9. Abschlußbetrachtungen

(Die Indizes im Gesamtüberblick)

Es bleibt festzuhalten, dass alle Indizes ihre Vor- und Nachteile haben, so dass je nach Einsatzgebiet ein anderer die optimale Wahl darstellt.
So ist APEX für feste Datenbanken, mit einem klar definierten Workload besonders effizient, da hier die mangelnden Updatefähigkeiten weniger gewichtig sind und die Performance optimal ist. Zudem ist die Datenmenge eher gering.
Sind dagegen inkrementelle Updates besonders wichtig kommen eigentlich nur die Indizes des Index Definition Schemes und die Index Fabric in Frage. Dabei liegt der weitere Schwerpunkt Ersterer bei einer möglichst geringen Datenmenge, während der Zweite besonders effiziente Plattenzugriffe ermöglicht.
Sowohl XISS als auch die „XPath Location“ Algorithmen sind nur eingeschränkt updatefähig, da hier die festen Pre- und Postorder-Ränge entscheidend sind. Um diese Indizes überhaupt ohne Neuerstellen aktualisieren zu können, müssen in der Pre- und Postorderreihenfolge Lücken gelassen werden und somit das Neuerstellen auf Fälle beschränkt werden kann, in denen die Lücken nicht mehr ausreichen.
Dabei sind sich diese Beiden, trotz der Unterschiede in der eigentlichen Suche, von den Grundlagen und ihrer Effektivität her ähnlich.
Die verschiedenen Indizes des Index Definition Schemes unterscheiden sich vor allem in Ihrer Genauigkeit und der Datenmenge, wobei die beiden Rangfolgen fast schon trivialerweise genau entgegengesetzt sind, so bietet der Forward-Backward-Index zwar die größte Präzision, benötigt aber auch am meisten Platz von den dreien. Genau umgekehrt ist es beim A(k)-Index falls das k genügend klein gewählt wurde, während sich der 1-Index immer in der Mitte präsentiert.
Gemeinsam haben die verschiedenen Indextypen, dass es im Detail schwierig sein kann sie anzupassen. So kann die Index Fabric zwar manuell optimiert werden, jedoch setzt dies eine sehr große Kenntnis des Administrators über die Daten und Anfragen voraus, die in der Regel nicht vorliegt. Ebenso schwierig ist es die Variablen des IDS zu bestimmen. Bei APEX wiederum ist nicht klar, ab wann ein Pfad als häufig eingeordnet werden kann und wann ein Update immer erforderlich bzw. nützlich ist, während bei XISS Extremfälle auftreten können, bei denen die Algorithmen versagen, also entweder sehr langsam oder im schlimmsten Fall falsch arbeiten.
Der direkten Anwendung am nächsten ist der Vorschlag von Thorsten Grust, da er sich sehr stark auf bereits eingesetzte Techniken beruft, allerdings ist der große Nachteil dieses Algorithmus sein großer Platzbedarf und seine mangelhafte Anpassbarkeit.

Name:	Index Fabric	Index Definition Scheme (General)	1-Index	A(k)-Index	Forward-Backward-Index	XISS Indexing and Storage	„XPath Location“	APEX
Paper:	A Fast Index for Semistructured Data	Covering Indexes for Branching Path Queries Updates for Structure Indexes				Indexing and Querying XML Data	Accelerating XPath Location Steps	APEX: An Adaptive Path Index for XML data
Scalability:	Scalable through prefix selection, further manual optimization for collected databases	Scalable by selecting suitable index	Medium precision	Lowest precision	Highest precision	Not scalable but suitable for any kind of database	not scalable but suitable for any kind of database	Scalable regarding to actual workload, best performance on collected databases
Update:	Incremental update possible (inserting new documents, changes by deleting and inserting new leaf)	Incremental update possible	Possible and efficient algorithm exists	Possible but no efficient algorithm yet	Possible and efficient algorithm exists	While there are gaps, else delete and rebuilding	Delete and rebuild	Update algorithm optimizes database according to the workload, so delete and rebuild
Efficiency:	HD loading (huge databases)	Depends on the used algorithm and configuration	arbitrary path expressions	arbitrary path expressions	arbitrary path expressions	reachability queries	reachablility queries	label path
Supported Queries:	Full XPath support		arbitrary path queries	arbitrary path queries	arbitrary path queries, Additional conditions	Full XPath support	Full XPath support, XQuery support	arbitrary path queries
Space:	High, as every Nodes and additional layers are saved	rather low due to extent building	medium	smallest	biggest	High as every node is saved with additional info	High as every node is saved with additional info	rather low due to extent building Usually smaller than 1-Index
Expense:	rather low	would be high but can be reduced very much	would be high but can be reduced by approaches	depends on k	High	building linear, fast look up but extreme cases	binear building, fast lookup	high while building, fast for label paths