Universität Kaiserslautern

Seminar der Arbeitsgruppe

Datenbanken und Informationssysteme

im Wintersemester 2002/2003

DB-Aspekte des E-Commerce,
Schwerpunkt: XML und Datenbanken

http://wwwdbis.informatik.uni-kl.de/courses/seminar

Indexstrukturen in XML

07.02.2003

Vanessa Schäfer

v_schaef@informatik.uni-kl.de

Inhalt

Abbildungen

Tabellen

1 Einführung

1.1 Indexstrukturen

Was ist ein Index?
Ein Index ist eine Datenstruktur, welche die Stelle, an der der indexierte Wert auftritt, identifiziert. Für Datenbanken bedeutet das, welcher Eintrag welchen Wert enthält. Zu jedem Index gehört ein Auswertungsalgorithmus, der diese Information verwendet, und Update-Algorithmen, die den Index warten. Zweck von Indexstrukturen ist die Beschleunigung der Auswertung von Anfragen, die an das Datenbanksystem gestellt werden. Sie ermöglichen direkten Zugriff auf Daten, ohne sequentiellen Zugriff auf die Datenbank. Allerdings können Indizes einen sehr großen Platzbedarf haben, es ist daher ein Kompromiss zwischen Speicherbedarf und Zeitersparnis zu treffen. Da beim Benutzen des Index kein Datenbank-zugriff erfolgt, muss bei dynamischen Datenbanken die Konsistenz zwischen Daten und Index erhalten bleiben. Daher sind regelmäßige Updates erforderlich, die im Idealfall nur die geänderten Daten betreffen, bei manchen Indizes allerdings eine Neuerstellung des Index erfordern.

Wo wird der Index eingesetzt?

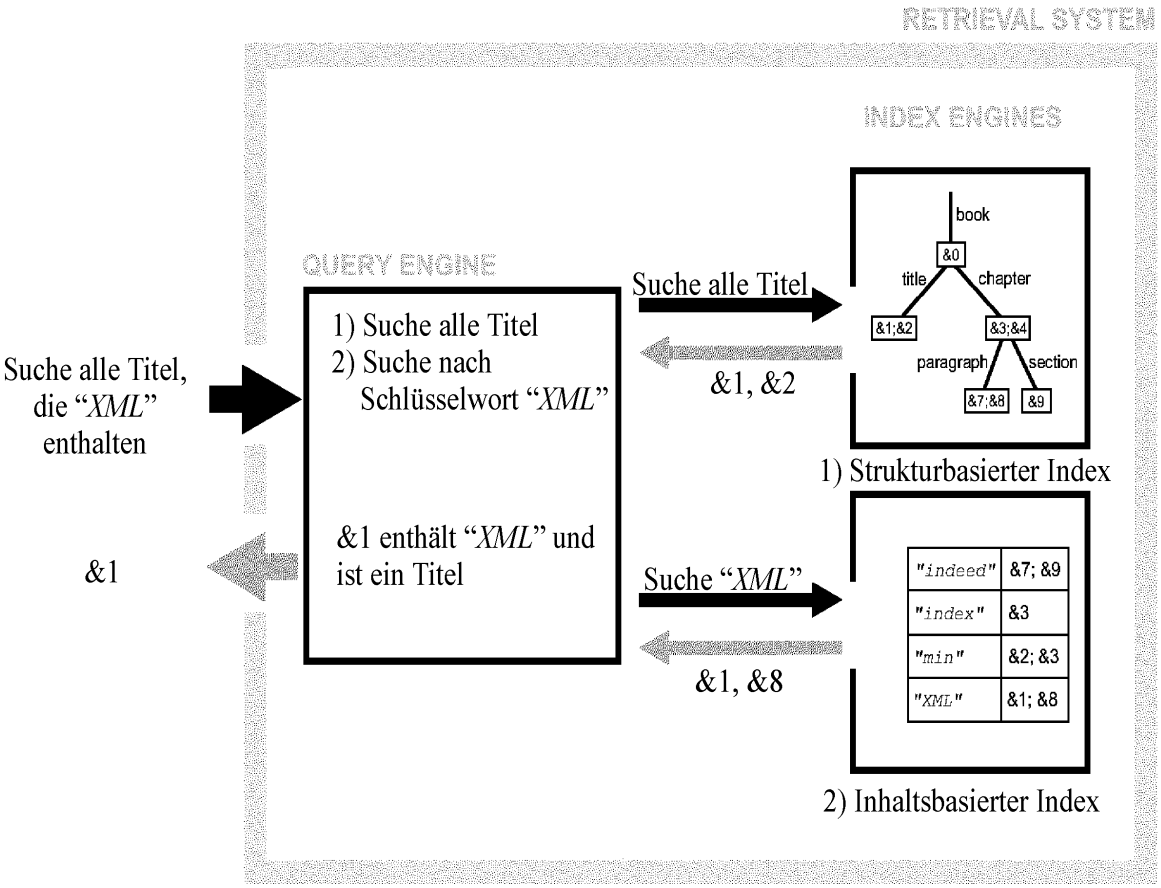


Abbildung 1.1: Architektur eines Retrieval Systems

Ein Retrieval System besteht aus zwei Typen von Komponenten: Eine Query-Engine und eine Menge von Index-Engines. Die Query-Engine bearbeitet die vom Benutzer gestellten Queries und erstellt Query-Plans, durch die die Index-Lookups ausgelöst werden. Jede Index-Engine bietet ein Interface an, in dem festgelegt ist, welche Typen von Lookups sie verarbeiten kann. Lookup-Patterns beschreiben das Interface bezüglich Input und Output. In der Query-Engine werden anhand dieser Lookup-Patterns eine oder mehrere Lookup-Queries erstellt, die von der Index-Engine ausgeführt werden. Abbildung 1.1 zeigt ein Beispiel für einen strukturbasierten (Kapitel 2.3) und einen inhaltsbasierten Index (Kapitel 2.2). Die Anfrage „Suche alle Titel, die „*XML*“ enthalten“, wird von der Query-Engine in zwei Lookup-Queries transformiert. Der strukturbasierte Index beantwortet die Query „Suche alle Titel“, der inhaltsbasierte die Query „Suche alle Einträge, die „*XML*“ enthalten“. Die Query-Engine führt die Ergebnisse beider Lookup-Queries zusammen und liefert „&1“ als Ergebnis zurück.

Exakte vs. Unexakte Indizes

Indizes lassen sich in exakte und unexakte Indizes unterteilen. Intuitiv ist ein Index exakt, wenn er genau die relevanten Daten für eine gegebene Query ausgibt und keine anderen.
Genauer lässt sich dies mit Hilfe von Precision und Recall beschreiben: Ein exakter Index hat optimale Precision und Recall. Sei D = Menge der Dokumente, O = Teilmenge der Dokumente, die zur Query passen und R = Menge der Treffer, die vom System zurückgeliefert werden. D, O und R dienen zur Definition von Precision und Recall und werden in Abbildung 1.2 schematisch dargestellt.

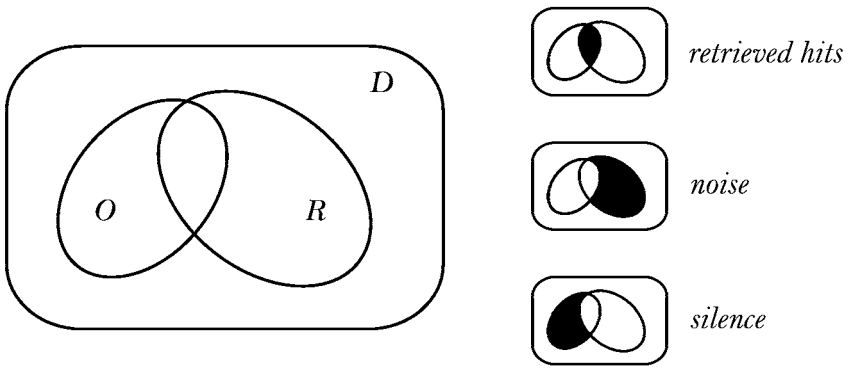


Abbildung 1.2: Precision und Recall

Definition Precision: Verhältnis von relevanten Suchergebnissen zu allen Suchergebnissen

$$= \frac{|O \cap R|}{|R|}$$

Definition Recall: Verhältnis von relevanten Suchergebnissen zu allen Dokumenten, die zu der Query passen

$$= \frac{|O \cap R|}{|O|}$$

Wichtig sind für die meisten Applikationen Indizes mit optimalem Recall, aber dafür eventuell nicht optimaler Precision (konservative Indizes). Progressive Indizes (volle Precision, dafür partieller Recall) werden bei gewichtetem Retrieval verwendet, um niedrig bewertete Ergebnisse auszuschließen. Ergebnisse unexakter Indizes müssen von der Query-Engine verifiziert werden.

1.2 Indexstrukturen und XML

XML-Daten gehören zu den teilweise strukturierten Daten (semistructured data), d. h., sie enthalten nicht nur eine Sequenz von Worten wie im klassischen Information Retrieval, sondern besitzen auch eine Struktur. Diese folgt aber nicht wie im Objekt-orientiertem Ansatz einem festen Schema. Abbildung 1.3 zeigt einen Ausschnitt aus einer XML- Datei, dargestellt als OEM-Graph. OEM (Object Exchange Model) dient speziell zur Darstellung teilweise strukturierter Daten als endlicher, beschrifteter und gerichteter Graph. Es wird Unvollständigkeit sowie Heterogenität in Typ und Struktur unterstützt.

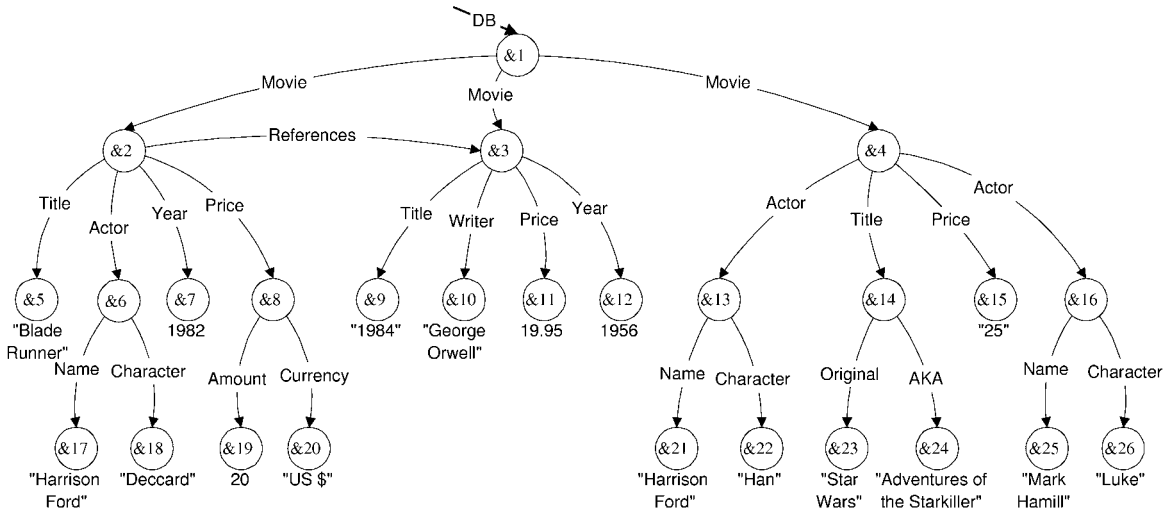


Abbildung 1.3: Ausschnitt einer OEM-Datenbank

Die Abbildung 1.3 zeigt einen Ausschnitt einer Video-Datenbank.
Vor allem die Heterogenität stellt ein Problem bei der Indexierung dar. So liefert zum Beispiel die Anfrage: „Suche alle Preise“

a) Einen Real: 19.95
b) Einen String: “25”
c) Und ein komplexes Objekt.

Indexstrukturen müssen also nicht nur mit Struktur ohne festem Schema umgehen können, sondern sollten auch Typumwandlung beherrschen.

1.3 Kriterien für Auswahl und Vergleich von Indexstrukturen

In [ZMR96] werden Richtlinien für die Präsentation und den Vergleich von Indexing-Techniken spezifiziert. Es folgt eine Zusammenfassung der wichtigsten Kriterien.
Das wichtigste Kriterium für Indizes ist wohl die *Anwendbarkeit*: Welche Typen von Anfragen werden von dem Index unterstützt? Jedes Index-Schema unterstützt nur eine Teilmenge aller möglichen Anfragen. Es gibt auch Klassen von Anfragen, die nur durch Scan der Datenbank beantwortet werden können.
Bei der Bewertung ist auch die *Erweiterbarkeit* von Interesse. Ein Index-Schema, das auf andere Typen erweitert werden kann, ist wertvoller als ein Schema, das in anderen Aspekten vielleicht sogar besser ist. Jeder zusätzliche Index, der in einem Datenbank-System zum Einsatz kommt, bedeutet zusätzlichen Platz- und Zeitbedarf.
Aufgrund stark anwachsender Speicherkapazitäten und Zunahme der zu speichernden Datenmengen ist auch die *Skalierbarkeit* von Interesse. Kann die Indexstruktur auch mit 100-facher Datenmenge effizient umgehen, oder ist dann eine andere Indexstruktur notwendig?
Auch die *Auswertungszeit* der Anfragen ist ein wichtiger Aspekt. Diese ist jedoch schwierig zu bewerten, da sie u. a. von CPU-Geschwindigkeit, Geschwindigkeit des Sekundärspeichers, Speichergröße und Systemauslastung abhängig ist.
Wichtig für dynamische Datenbanken ist das Verhalten bei Updates. Manche Indizes unterstützen inkrementelle Updates und erneuern die von der Änderung betroffenen Ausschnitte, andere bauen jedoch unter bestimmten Umständen den Index komplett neu auf.

Weitere Aspekte:

- Wie viel Speicherplatz benötigt der Index?
- Wie lange dauert die Konstruktion des Indizes und wie viel Platz benötigt er währenddessen ?
- Wie lange dauert ein Update?
- Welche Datenstruktur wird verwendet, um den Index zu speichern?

Messzahlen für die Bewertung von Indizes:

Retrieval Latency: Zeitbedarf für die Bearbeitung einer Anfrage
Storage Overhead: Verhältnis von Index-Größe zu Datenbankgröße

$$= \frac{index\ size}{database\ size}$$

Throughput: Anzahl der Datenbankobjekte, die in einer bestimmten Zeit indexiert werden können

$$= \frac{number\ of\ indexed\ nodes\ or\ links\ in\ the\ database}{time\ unit}$$

2 Indexstrukturen in XML

2.1 Überblick

Indexstrukturen in XML lassen sich in inhaltsbasierte Indizes, strukturbasierte Indizes und hybride Indexstrukturen für textlichen und strukturellen Inhalt unterscheiden. Inhaltsbasierte Indizes stammen aus dem klassischen Information Retrieval. Sie betrachten nur den textlichen Inhalt eines Dokumentes, ohne dessen Struktur zu beachten. Beispiele sind Text Index und Value Index. Strukturbasierte Indizes konzentrieren sich auf die Struktur des Dokumentes. Vertreter dieser Klasse sind DataGuide und T-Index. In Datenbanksystemen müssen Indizes beider Klassen parallel eingesetzt werden. Diese werden dann für Anfragen, die Inhalt und Struktur betreffen, kombiniert. Um sich die Notwendigkeit mehrerer Indexstrukturen in einem Datenbanksystem zu ersparen, hat man Indizes entwickelt, die sowohl textlichen Inhalt als auch Struktur umfassen. Als Beispiele sind hier IndexFabric und BUS zu nennen. Eine besondere Indexstruktur stellt SphinX dar: hier werden bei DTD- konformen Dokumenten die Schema-Informationen benutzt, um die Geschwindigkeit zu erhöhen.

2.2 Inhaltsbasierte Indexstrukturen

Inverted Files
Inverted Files [GBYS91], [ZMR98] stammen aus dem klassischen Information Retrieval und entsprechen dem Buch-Index. Für jedes Schlüsselwort wird eine Liste von Referenzen auf Vorkommnisse im Text gespeichert. Beim Lesen einer Datei / eines Buches wird genau umgekehrt vorgegangen: man sieht auf eine Seite und liest die darauf stehenden Wörter. Daher der Name Inverted Files. Inverted Files bestehen aus zwei Teilen: Einem Vokabular, das alle zu indexierenden Werte enthält und zu jedem Wert eine invertierte Liste. Vokabulare sollten in Form von B^+-Bäumen gespeichert werden, deren Blätter Pointer auf die invertierten Listen enthalten. Ähnlich gute Ergebnisse zeigen aber auch Hash Tables und Arrays. Um eine gute Performanz zu ermöglichen, sollten die Inverted Lists in sortierter Form gespeichert werden.
Im Folgenden werden zwei Indexstrukturen vorgestellt, die auf Inverted Files basieren. Tabelle 1 zeigt die wichtigsten Merkmale von Text Index und Value Index.

Index	Text Index	Value Index
Input → Output	Keywords → NodeIDs	Predicates → NodeIDs
Indexstruktur	Inverted File	Inverted File
DB-Struktur	Graph	Graph
Inkrementelle Updates	Ja	Ja
Speicherbedarf	5-15% Storage Overhead (Angabe für Inverted Files)	5-15% Storage Overhead (Angabe für Inverted Files)
Einschränkungen	Keine Struktur	Keine Struktur
Besonderheiten		Kann Typecasting unterstützen

Tabelle 1: Eigenschaften von Text Index und Value Index

2.2.1 Text Index

Der Text Index bildet ein gegebenes Schlüsselwort auf alle Dokumentknoten, die es enthalten, ab. Text Indizes kann man auf direkten Inhalt beschränken, d. h., es werden nur die tiefsten Knoten des Graphen betrachtet. Wird indirekter Inhalt unterstützt, speichert der Index das Vorkommen des Schlüsselworts für jeden Knoten des betroffenen Pfades bis zur Wurzel. Der erste Fall bietet zwar eine Platzersparnis, dafür muss aber die Query-Engine überprüfen, ob es innere Dokumentknoten gibt, die das gewünschte Schlüsselwort indirekt enthalten. Allerdings unterstützt der Text Index keinen strukturellen Inhalt. Es müssen auf jeden Fall andere Indexstrukturen verwendet werden, um den strukturellen Teil der Anfrage zu beantworten.
Es gibt für Text Indizes verschiedene Möglichkeiten zur Reduzierung der Menge der Schlüsselwörter. Eine Erweiterungsmöglichkeit ist die Unterstützung von Stemming. Hierzu werden alle Schlüsselwörter durch ihren Wortstamm ersetzt, z. B. „to index“ und „indexing“ durch „index“. Im englischen sind die Algorithmen dazu relativ einfach, da hier die meisten Wörter ihren Stamm als Präfix enthalten. Eine andere Möglichkeit ist die Verwendung von Stoppwortlisten. Sie enthalten Wörter wie Präpositionen und Artikel, aber auch einfach besonders häufig vorkommende Wörter. Diese werden nicht indexiert, da sie so oft vorkommen, dass sie die Suche fast nicht eingrenzen. Ein dritter Ansatz ist die explizite Auswahl der zu indexierenden Wörter. Zumindest die beiden letzten Ansätze sind für jede betroffene Datenbank einzeln vom Administrator festzulegen, da sie eine gewisse Kenntnis der gespeicherten Daten und der häufig gestellten Anfragen voraussetzen.

Tindex in Lore
Lore (Leightweight Object REpository) [Wid99] ist ein Datenbankmanagementsystem, das in Stanford entwickelt wurde. Ursprünglich war es für teilweise strukturierte Daten ausgelegt, wurde aber später für XML modifiziert. In Lore wird unter anderem ein Text Index (hier Tindex genannt) [MWA$^+$ 98] verwendet. Tindex liefert zur Eingabe von Wort w und Label l als Ausgabe eine Menge von Tupeln (o, n) mit Objekt o, in dem w an der n- ten Stelle vorkommt. Gespeichert werden die invertierten Listen in Form von Hash-Tabellen.

Beispielanfrage zu Abbildung 1.3 (Seite 6):
Suche alle Namen die „Ford“ enthalten.
Ausgabe: {(&17,2), (&21,2)}

2.2.2 Value Index

Ein Value Index ist prinzipiell eine Generalisierung eines Text Index. Er unterstützt verschiedene Selektionskriterien zu verschiedenen Datentypen. Selektionskriterien sind z. B. für Zahlen „>“, „<“ oder „=“, für Strings „contains“. Wie der Text Index kann auch der Value Index auf direkten Inhalt begrenzt werden.

Vindex in Lore
Der Vindex [MWA$^+$ 98] ist eine in Lore implementierte Variante des Value Index. Er unterstützt die Datentypen Integer, Real und String. Für jedes indexierte Label gibt es drei verschiedene Indizes, die als B$^+$- Bäume gespeichert werden.
- Jss indexiert alle Strings,
- Jsr indexiert alle Strings, die in Reals verwandelt werden können als Reals,
- Jrr indexiert alle Reals und alle Integer als Reals.

Dies ermöglicht bei Suche des Wertes „5“ sowohl nach dem Integer „5“ als auch nach dem Real „5.0“ und den Strings „5“ und „05“ zu suchen.
Beispielanfrage zu Abbildung 1.3 (Seite 6):
Suche DB.Movie.Price mit DB.Movie.Price>15
Ausgabe: {&11, &15}

2.2 Strukturbasierte Indexstrukturen

Index	DataGuide	T-Index
Input → Output	Simple Paths→ NodeIDs	1-Index: Simple Paths→ NodeIDs 2-Index: Relative Paths→ NodeIDs T-Index: Privileged Paths→ NodeIDs
Indexstruktur	Graph	Graph
DB-Struktur	Graph	Graph
Inkrementelle Up-dates	Ja	Ja
Speicherbedarf	Worst Case: exponentiell Im Experiment je nach DB-Struktur 2-317% Storage Overhead	Worst Case: 1-Index linear 2-Index quadratisch
Einschränkungen	Keine relativen Pfade, kein Inhalt	
Besonderheiten		Path Templates

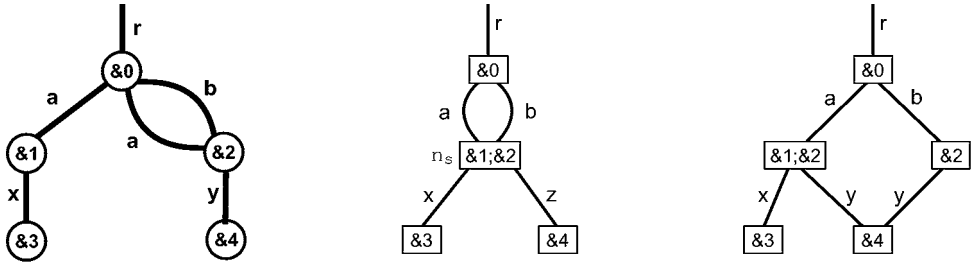
Tabelle 2: Eigenschaften von DataGuide und T-Index

2.3.1 DataGuide

Ein DataGuide [GW97] für eine gegebene Datenbank ist ein Graph, in dem jeder einfache Pfad der Datenbank genau einmal vorkommt. Für einen exakten Index benötigt man noch zusätzlich die Beschränkung, dass keine Pfade auftreten dürfen, die nicht in der Datenbank auftreten. Dies wird durch strong DataGuides erfüllt.
Definition strong DataGuide:

> Sei s ein Datenbankobjekt und d der zugehörige DataGuide. Weiter sei l ein Label Path von s, Ts(l) die Zielmenge von l in s und Td(l) die Zielmenge von l in d. Sei Ls(l)={m|Ts (m)=Ts(l)}, also Ls(l) ist die Menge aller Label Paths in s, die die gleiche Zielmenge wie l haben. Analog sei Ld(l)={m|Td(m)=Td(l)}.
> Gilt für alle l in s Ls(l)=Ld(l), dann ist d ein strong DataGuide.

Des Weiteren enthält jedes Objekt des DataGuides einen Link auf die Zielmenge des zugehörigen DB-Objekts. Abbildung 2.1 zeigt einen weak DataGuide und einen strong DataGuide zu einem Datenbank-Graphen.



a) Datenbank-Graph b)weak DataGuide c) strong DataGuide
Abbildung 2.1: DataGuide vs. Strong DataGuide

Pindex in Lore
In Lore kommt neben Tindex und Vindex auch ein Pindex (Path Index) zum Einsatz. Er dient zur Bearbeitung des strukturellen Teils der Anfrage und wird durch DataGuides realisiert. Ein Pindex–Lookup für einen Pfad p gibt eine Menge von Objekten zurück, die über p erreichbar sind. Zur Zeit werden nur Anfragen mit einfachen Pfadausdrücken (ohne Wildcards) unterstützt und Wertevergleich ist nur gegen Konstanten möglich.
Beispielanfrage zu Abbildung 1.3 (Seite 6):
Suche DB.Movie.Title
Ausgabe: {&5, &9, &14}

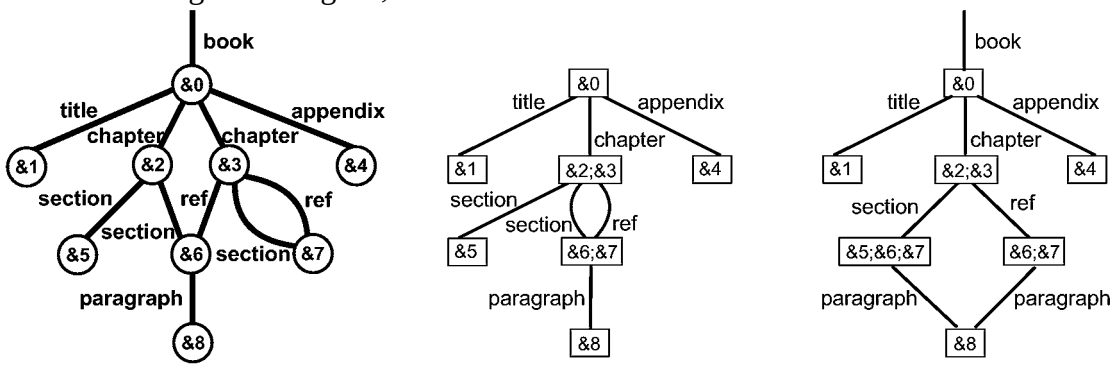
2.3.2 T-Index

Der T-Index (Template Index) [MS99] ist ein struktureller Index mit den Spezialfällen 1-Index und 2-Index. Die Klasse der Pfade, die zu einem gegebenen T-Index passen, werden durch ein Path Template beschrieben.

Path Templates
Ein Path Template t hat die Form $T_1x_1T_2x_2...T_nx_n$, mit T_i= {regulärer Pfadausdruck | [P] | [F]}. [P] steht für einen beliebigen regulären Pfadausdruck und [F] für eine Formel. Inst(t) ist die Menge der Instanzen von t. Für das Path Template (-*.Restaurant)x_1[P]x_2Name x_3 [F]x_4
ist der Query Path q= (-*.Restaurant)x_1* x_2 Name x_3 Friday x_4 eine mögliche Instanz. Ziel ist es, zu jedem spezifizierten Path Template t einen Index zu konstruieren, der die Anfragen in inst(t) unterstützt.

1-Index
Der 1-Index unterstützt Anfragen $\in$ inst ([P] x), also reguläre Abfragen. Damit ist der Index nicht auf eine spezielle Klasse von Queries optimiert und alle von der Wurzel ausgehenden Label Paths in der Datenbank werden indexiert. Jeder Knoten des Index referenziert alle Dokumentknoten, die über den gleichen Label Path wie der Knoten im Index erreichbar sind. Abbildung 2.2 zeigt einen Datenbank-Graphen (a), den zugehörigen 1-Index (b), und den zugehörigen DataGuide (c). Unterschiede zum DataGuide sind:
- das Label der Wurzel wird weggelassen,
- Dokumentknoten werden genau einmal referenziert und
- Der Pfad /chapter/section kommt zweimal vor.

Für baumförmige Datenbanken sind 1-Index und DataGuide identisch. Die Evaluierung von relativen Anfragen ist möglich, kann aber einen vollen Index-Scan auslösen.



a) Datenbank-Graph b) 1-Index c) strong DataGuide
Abbildung 2.2: 1-Index vs. DataGuide

2-Index

Ein 2-Index unterstützt Anfragen $\in$ inst(-* x_0 [P] x_1), d. h., es werden sowohl absolute als auch relative Pfade indexiert. Im Prinzip extrahiert der 2-Index alle Paare von Knoten, die über einen gegebenen Label Path verbunden sind. Abbildung 2.3 zeigt einen 2-Index für die Datenbank aus Abbildung 2.2 (a). In den Knoten, die mit p0 bis p4 markiert sind, entspricht der Teil -* der Templates dem leeren Pfad; es sind absolute Pfade. Für p5 bis p8 entspricht -* /chapter und für p9 /chapter/section und /chapter/ref (relative Pfade).

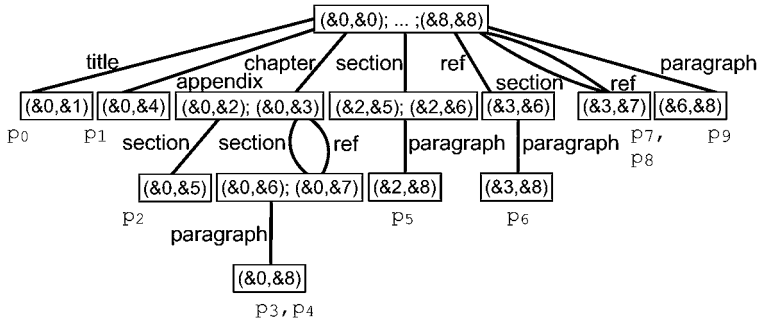
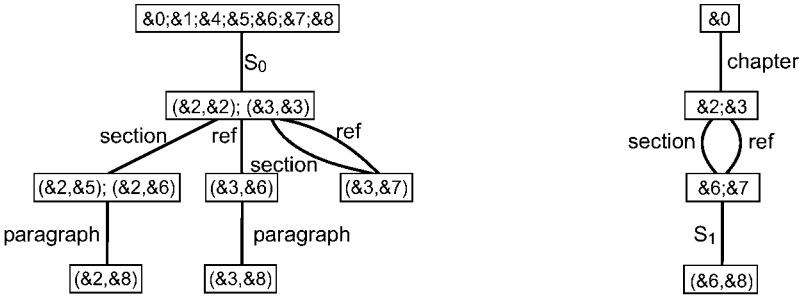


Abbildung 2.3: 2-Index

T-Index

Der T-Index ist der allgemeinste der drei Indizes, er akzeptiert alle Path Templates der Form t = $T_1x_1T_2x_2...$ T_nx_n. Im Gegensatz zu 1- und 2-Index enthalten die Knoten Tupel mit 1 bis k NodeIDs, wobei nur k-Tupel Treffer sind. Die Knoten enthalten jeweils die Tupel, bei denen für alle Paare von aufeinanderfolgenden Variablen im Template die zugehörigen Paare von Dokumentknoten des Tupels über die gleiche Menge von Label Paths verbunden sind.

Abbildung 2.4 zeigt zwei vereinfachte T-Indizes für Abbildung 2.2 (a). In Abbildung 2.4 (a) repräsentiert die Kante S0 den regulären Pfadausdruck /chapter, in (b) steht S1 für /paragraph. In (b) wird ein Lookup der Form /chapter/section/paragraph übersetzt zu /chapter/section/S1 und liefert(&6, &8). Dies bedeutet, dass &6 von der Wurzel aus über /chapter/section erreicht werden kann und &8 von &6 aus über /paragraph.



a) T-Index für /chapter x_0 [P] x_1 b) T-Index für [P] x_0 paragraph x_1

Abbildung 2.4: T-Indizes für verschiedene Path Templates (vereinfacht)

2.4 Hybride Indexstrukturen

Index	IndexFabric	BUS
Input → Output	Simple Paths → NodeIDs Privileged Paths → NodeIDs	Simple Paths x Keywords → DocIDs x NodeIDs x Weights
Indexstruktur	Baum	Baum
DB-Struktur	Baum	Baum
Inkrementelle Updates	Ja	Nur teilweise (MBM: Ja)
Speicherbedarf	In Experimenten bis 150% Storage Overhead	In Experimenten bis 240% Storage Overhead
Besonderheiten	Refined Paths	Ranking der Treffer

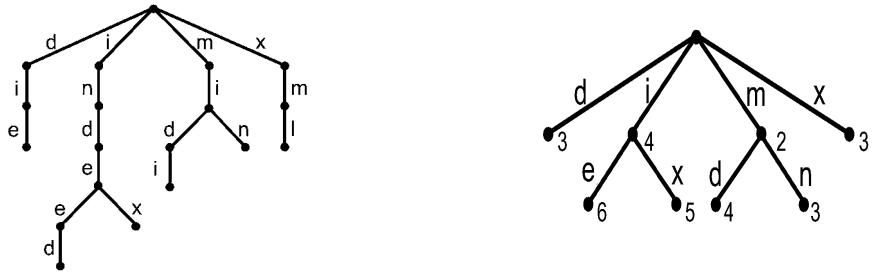
Tabelle 3: Eigenschaften von IndexFabric und BUS

2.4.1 IndexFabric

Aufbau

IndexFabric [CSF+01] konvertiert Pfade in Strings und nutzt als Basis Patricia Tries zur kompakten und effizienten Speicherung. Ein Trie ist ein beschrifteter Baum zur Indexierung von Strings. Jede Kante ist mit einem Symbol beschriftet, der ganze String lässt sich als Folge der Kanten von Wurzel bis Blatt rekonstruieren. PATRICIA Tries erweitern Tries um Kompression. Es werden alle Kanten weggelassen, an denen sich das Schlüsselwort an der betroffenen Position nicht von einem anderen Schlüsselwort unterscheidet.

Abbildung 2.5 zeigt einen Trie und einen PATRICIA Trie für die Schlüsselwörter „die“, „indeed“, „index“, „midi“, „min“ und „xml“. Die Zahlen in Abbildung 2.5 b) stehen für die Anzahl der Zeichen bis zu diesem Knoten.



a) Trie b) PATRICIA Trie

Abbildung 2.5: Trie und PATRICIA Trie

Im Folgenden wird nur noch von Tries gesprochen, obwohl PATRICIA Tries zum Einsatz kommen.

Um den Speicherzugriff zu optimieren, wird mit Hilfe der Tries ein Schichten- Modell realisiert, wie es in Abbildung 2.6 dargestellt wird. Der Basis-Trie wird in Blöcke passend zur Größe der Speicherblöcke aufgeteilt. Diese Blöcke werden über die Präfixe der Schlüsselworte durch einen Trie der nächsthöheren Schicht indexiert. Passt der Trie der höchsten Schicht nicht mehr in einen Speicherblock, wird er aufgeteilt und eine neue Schicht hinzugefügt.

Zwischen den Schichten gibt es zwei Sorten von Links:

- Far Links (in Abbildung 2.6 durch → dargestellt), die normalen, beschrifteten Kanten entsprechen

- Direct Links (durch - -> dargestellt), die einen Knoten mit dem Block der darunter liegenden Schicht verbinden, dessen Wurzel das gleiche Präfix repräsentiert.

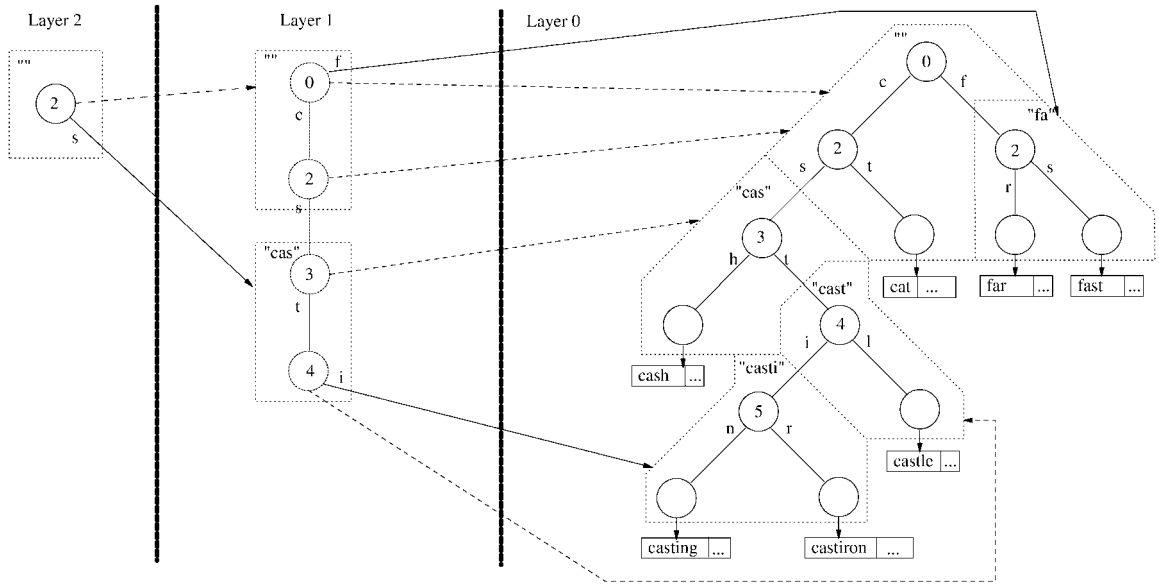


Abbildung 2.6: Layered IndexFabric

Suche
Die Suche beginnt in der höchsten Schicht und traversiert die Schichten entsprechend dem Label der Kanten bzw. Links. Pro Schicht muss ein Block untersucht werden. Durch die verlustbehaftete Kompression der Tries kann aber der falsche Block ausgewählt werden, was Backtracking notwendig macht. Außerdem müssen die gefundenen Treffer verifiziert werden. IndexFabric speichert die Suchschlüssel sehr kompakt. Für eine Billion Schlüssel genügen drei Schichten, von denen nur die unterste so groß ist, dass sie auf Platte gespeichert werden muss. In diesem Fall genügt also maximal ein I/O Zugriff pro Suchschlüssel.

Aktualisieren
Aktualisieren, Einfügen und Löschen können sehr effizient durchgeführt werden. Einfügen bewirkt im schlimmsten Fall das Hinzufügen eines Knotens pro Schicht und das Erstellen einer neuen Schicht. Im Normalfall wird allerdings nur ein weiterer Knoten auf der untersten Ebene hinzugefügt. Zum Löschen wird der betroffene Schlüssel gesucht und die entsprechende Kante entfernt. Aktualisierung erfolgt durch Löschen und anschließendes Einfügen.

Indexieren von XML
Zum Indexieren werden die XML-Pfade mit Hilfe von Bezeichnern (designator) als Schlüssel kodiert. Jedes Tag wird eindeutig durch ein Zeichen oder eine Zeichenkette repräsentiert. So steht „IBN ABC Corp“ in Abbildung 2.8 für

```
<invoice>
    <buyer>
        <name>
            ABC Corp
        </name>
    </buyer>
</invoice>
```

Bezeichner werden beim Erstellen des Index wie normale Zeichen behandelt.

Raw Paths indexieren die hierarchische Struktur von XML durch Kodieren der Pfade von der Wurzel bis zum Blatt als Strings. Auf diese Weise wird das komplette XML-Dokument indexiert. Abbildung 2.8 zeigt ein Beispiel für Raw Paths zu dem XML-Dokument aus Abbildung 2.7.

Doc 1:
```
<invoice>
  <buyer>
    <name>ABC Corp</name>
    <address>1 Industrial Way</address>
  </buyer>
  <seller>
    <name>Acme Inc</name>
    <address>2 Acme Rd.</address>
  </seller>
  <item count=3>saw</item>
  <item count=2>drill</item>
</invoice>
```

Doc 2:
```
<invoice>
  <buyer>
    <name>Oracle Inc</name>
    <phone>555-1212</phone>
  </buyer>
  <seller>
    <name>IBM Corp</name>
  </seller>
  <item>
    <count>4</count>
    <name>nail</name>
  </item>
</invoice>
```

Abbildung 2.7: XML Dokument

(a)
<invoice> = **I**
<buyer> = **B**
<name> = **N**
<address> = **A**
<seller> = **S**
<item> = **T**
<phone> = **P**
<count> = **C**
count (attribute) = **C'**

(b)

Document 1	Document 2
I B N ABC Corp	**I B N** Oracle Inc
I B A 1 Industrial Way	**I B P** 555-1212
I S N Acme Inc	**I S N** IBM Corp
I S A 2 Acme Rd.	**I T C** 4
I T drill	**I T N** nail
I T C' 2	
I T saw	
I T C' 3	

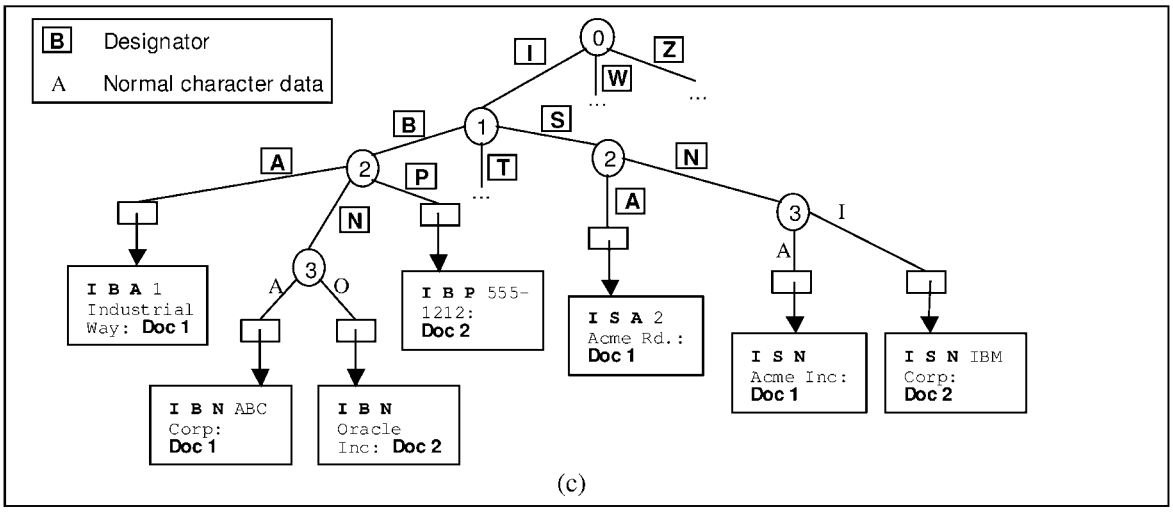


Abbildung 2.8: Raw Paths

Refined Paths sind spezielle Pfade, die häufige Zugriffsmuster optimieren. Sie werden ebenfalls durch Bezeichner identifiziert, ansonsten werden sie wie Raw Paths kodiert. Welche Anfragen durch Refined Paths optimiert werden, entscheidet der Administrator. Gespeichert werden sie im gleichen Index wie die Raw Paths.

2.4.2 BUS

BUS (Bottom-Up-Scheme) [OSS01] unterstützt komplexe Anfragen, die sowohl Inhalt als auch Struktur betreffen. Zusätzlich ist eine Bewertung der Treffer möglich; von allen Knoten, die zum strukturellen Teil passen, wird der am höchsten bewertet, in dem das Schlüsselwort am häufigsten vorkommt.

Struktur
Der BUS Index besteht aus drei Datenstrukturen, die in Abbildung 2.9 dargestellt werden.
Der Schema-Tree (a) dient als struktureller Index. Außerdem gibt es einen Content Index (b) und eine Akkumulator–Tabelle (c) die die Häufigkeit eines Schlüsselworts in den Dokumentknoten speichert.
Der Schema-Tree ist unabhängig von der Datenmenge, er gibt das Datenbank-Schema wieder.
Der Content Index ist ein Inverted File mit direktem textlichen Inhalt, d. h., es werden nur die untersten Knoten betrachtet. Zusätzlich werden noch Informationen gespeichert, um einen Join der inhaltlichen und strukturellen Treffer sowie das Ranking zu ermöglichen. Einträge haben die Form „Keyword“ → (docID, nodeID, level, type, frequency). Als konkretes Beispiel in Abbildung 2.9 bedeutet also „XML“ → (&25, &802, 3, #7, 4), dass im Knoten &802 von Dokument &25, der sich auf Level 3 befindet und vom Elementtyp #7 ist, das Schlüsselwort „XML“ genau vier Mal vorkommt. Für jeden Dokument-Knoten gibt es einen Eintrag in der Akkumulator –Tabelle.

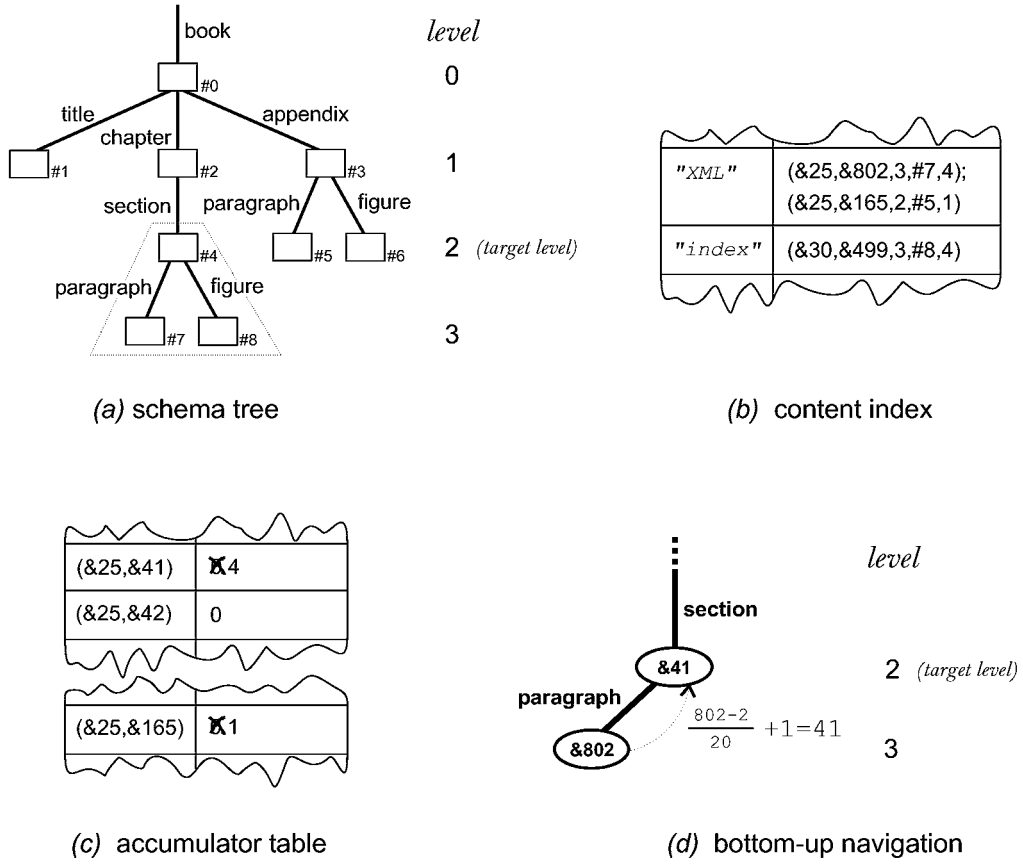


Abbildung 2.9: BUS Index

Beispiel eines Lookups
Das folgende Beispiel bezieht sich auf Abbildung 2.9
Anfrage: Suche /book/chapter/section[“XML”]
Zuerst werden aus dem Schema Tree die der geforderten Struktur entsprechenden Knoten extrahiert, hier der Knoten mit der Elementtyp-Nummer #4. Relevant für den Inhalt dieser Anfrage ist somit nur noch der Subbaum mit der Wurzel #4 und den Knoten #7 und #8. Diese werden in der Menge Types (={#7, #8 }) gespeichert. Außerdem wird der Level des zuerst extrahierten Knotens, genannt Target-Level gespeichert, hier also l_t=2. Nun wird der Context Index durchsucht. Ist die NodeID eines gefundenen Eintrags nicht in Types enthalten, so wird nur die Häufigkeit des Schlüsselwortes in der Accumulator-Tabelle gespeichert. Dies dient später zur Gewichtung der Resultate. Im anderen Fall muss die NodeID des Vorgängerknotens auf Ebene l_t berechnet werden. Dies geschieht durch die iterierte Anwendung der Formel

$$ParentID = \left[\frac{(ChildID - 2)}{a} + 1 \right]$$ mit a= maximale Anzahl der Kindknoten.

Für die so ermittelten Knoten wird ebenfalls ein Eintrag in der Akkumulator-Tabelle vorgenommen. Im Beispiel wird also der Eintrag zu (&25, &165) auf 1 gesetzt, da hier „XML“ einmal vorkommt. Des Weiteren wird der Vorgängerknoten auf Level l_t =2 von &802 berechnet. Mit a=20 ergibt sich &41. Also wird in der Akkumulator-Tabelle (&25, &41) auf 4 gesetzt. Das Ergebnis der Query ist (&25, &41, 4), d. h., „/book/chapter/section[„XML“]“ kommt im Dokument &25 an Knoten &41 vier Mal vor.

MBM
MBM (Modified BUS Method) ist eine Erweiterung von BUS mit dem Ziel, den Platz- und Zeitbedarf zu verringern. So werden nicht mehr alle Schlüsselwörter und nur noch bestimmte Elementtypen indexiert. Die setzt allerdings eine gewisse Kenntnis der zu indexierenden Daten voraus. Ein großer Nachteil von BUS ist, dass, sobald die maximale Anzahl der Kindknoten erreicht ist, eine komplette Neubildung beider Indizes notwendig wird, da sonst die Berechnung der ParentID nicht mehr korrekt durchgeführt werden kann. Bei MBM werden die hinzukommenden Knoten stattdessen fortlaufend nummeriert, zusätzlich wird aber in jedem Knoten die ParentID gespeichert. Somit sind inkrementelle Updates immer möglich.

2.5 SPHINX

Index	SphinX
Input → Output	Regular Path Expression → NodeIDs
Indexstruktur	Graph
DB-Struktur	Graph
Inkrementelle Updates	Ja
Speicherbedarf	160-300% Storage Overhead
Besonderheiten	DTD wird benutzt

Tabelle 4: Eigenschaften von SphinX

SphinX [PH01] unterscheidet sich von den bereits vorgestellten Indexstrukturen durch die Verwendung der Schema-Information DTD-konformer Dokumente, um die Indexierung zu beschleunigen. Zum Speichern indexierter Werte werden B^+-Bäume verwendet, eine in DB-Systemen standardmäßig angewandte und daher auch ausgereifte Technik (im Gegensatz zu

IndexFabric, wo noch nicht bekannt ist, was die Datenstruktur wirklich leisten kann). Tabelle 5 zeigt einen Vergleich der Eigenschaften von Lore, IndexFabric und SphinX.

Feature	Lore	IndexFabric	SphinX
DTD verwendet	Nein	Nein	Ja
Einzelner Index	Nein	Ja	Ja
Spezialisierte Indexstrukturen	Ja	Ja	Nein
Range Queries effizient	Ja	Nein	Ja
Getestet mit großen Dokumenten	Nein	Nein	Ja
Vergleich mit Lore	-	Nein	Ja

Tabelle 5: Vergleich von Indexstrukturen

SphinX unterstützt sowohl einfache Pfadausdrücke (absolute Pfade) als auch allgemeine Pfadausdrücke. Auch hier wird die Auswahl der zu indexierenden Pfade dem Administrator überlassen. Eingabe für SphinX ist eine Menge von DTDs und eine Menge von XML - Dokumenten, von denen jedes konform zu einer der DTDs ist. Zur Vereinfachung wird im Folgenden von einer DTD und einem XML-Dokument ausgegangen.
SphinX besteht aus drei Teilen: einem Dokument-Graph, der das XML-Dokument als Graph darstellt, einem Schema-Graph als Darstellung der DTD und einer Menge von B^+-Bäumen. Abbildung 2.10 zeigt eine DTD einer Datenbank und ein dazu konformes Fragment eines XML- Dokumentes. Abbildung 2.11 zeigt den zugehörigen Dokument-Graph und die Struktur des Index.

```
<!ELEMENT bib (book, article)* >
<!ELEMENT book (author+, title, publisher) >
<!ATTLIST book year CDATA >
<!ELEMENT article (author+, title, year? ) >
<!ELEMENT title CDATA>
<!ELEMENT year CDATA>
<!ATTLIST article conference CDATA >
<!ELEMENT publisher (name, address*) >
<!ELEMENT name CDATA>
<!ELEMENT address CDATA>
<!ELEMENT author (firstname?, lastname)>
<!ELEMENT firstname CDATA>
<!ELEMENT lastname CDATA>
```

```
<bib>
  <book year="1995">
    <title>An Introduction to Database Systems</title>
    <author> <lastname>Date </lastname> </author>
    <publisher> <name>Addison–Wesley </name>
    <address>Boston, MA 02116<address> </publisher>
  </book>
  <book year="2001">
    <title>Foundation for Object/Relational Databases</title>
    <author><lastname>Darwen </lastname> </author>
    <publisher><name> Addison–Wesley </name></publisher>
  </book>
  <article conference="ICDT">
    <author> <lastname>Ullman</lastname>
        <firstname> Jeffrey D.</firstname>
    </author>
    <title> Querying Websites Using Compact Skeleton </title>
    <year> 2001 </year>
  </article>
</bib>
```

Abbildung 2.10: Beispiel DTD und XML Fragment

Der Dokument-Graph
Der Dokument-Graph ist ein Wurzelgraph mit beschrifteten Knoten. Elemente und Attribute des XML-Dokuments bilden die Knoten, Eltern/Kind-Beziehungen werden durch Kanten dargestellt und Blätter stellen die atomaren Werte dar. Die Anordnung der Elemente im Originaldokument wird durch die Organisation der Subelemente in Reihenfolge ihres Auftretens erhalten. Der Graph ist bidirektional, um top-down und bottom-up Traversierung zu ermöglichen. Die Größe des Graphen ist eine Funktion in Abhängigkeit von der Größe des XML-Dokuments und daher einfach zu bestimmen.

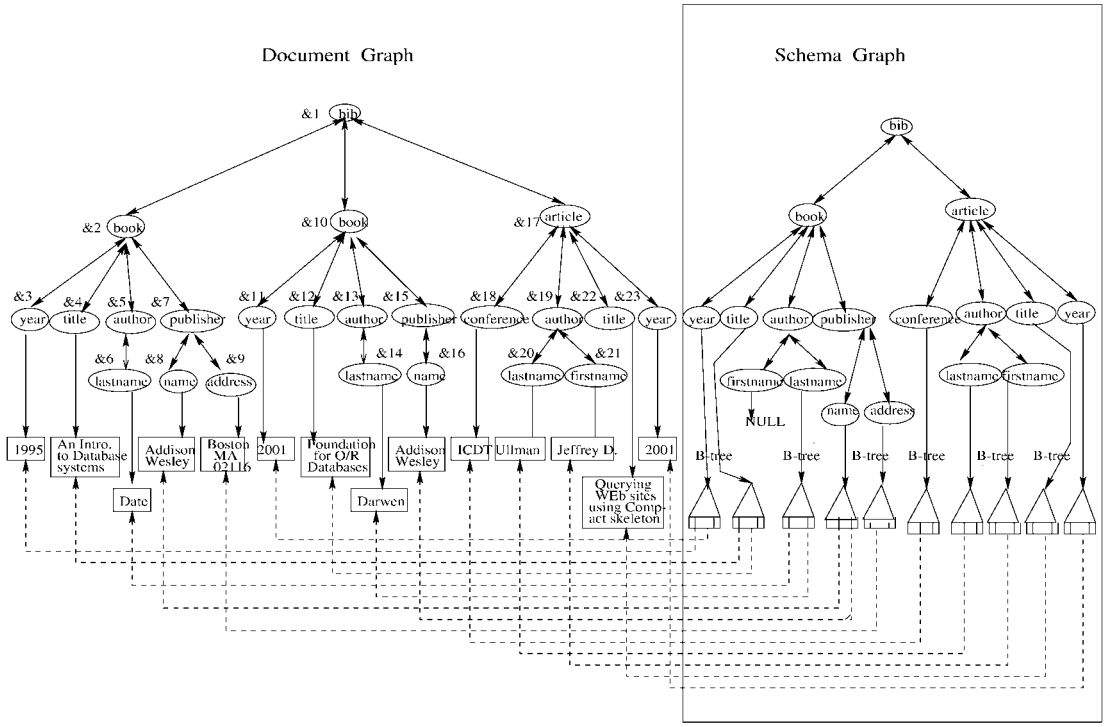


Abbildung 2.11: Beispiel Dokument-Graph und SphinX-Indexstrukturen

Der Schema-Graph
Der Schema-Graph dient zur Bestimmung relevanter Pfade im Dokument-Graph (path identification). Er ist ebenfalls ein bidirektionaler Wurzelgraph mit beschrifteten Knoten. Um einen Graphen endlicher Größe zu erhalten, werden Kardinalitäts-Restriktionen (?,+, *) ignoriert und Alternativen (a|b) durch Konjunktionen (a, b) ersetzt. Dabei bleibt allerdings die Vollständigkeit erhalten, d. h., DTD-konforme Dokumente sind gültig gegenüber dem vereinfachten Graphen. Außerdem kann der Graph in einer Zeitspanne proportional zur Anzahl der Elemente der DTD gebildet werden. Die Blätter des Graphen beinhalten Pointer auf einen B^+-Baum, falls der entsprechende Pfad indexiert wird, ansonsten Null-Pointer.

B^+-Bäume
Die B^+-Bäume stellen pfad-spezifische Indizes dar, entsprechend der atomaren Werte der Pfade im Dokument-Graph. Da in DTDs Daten als Character-Strings dargestellt werden, bestehen die B^+-Bäume aus Strings. Die B^+-Bäume enthalten Pointer auf die entsprechenden Knoten des Dokument-Graphen.

Anfrageverarbeitung
Durch den Index-Path-Extraction-Algorithmus werden die (allgemeinen) Pfadausdrücke aus der Anfrage in eine Menge von absoluten Pfaden verwandelt und über den Schema-Graph die indexierten Pfade ermittelt. Als Ausgabe werden eine Menge von absoluten Pfaden sowie die Pointer auf die Wurzel der zugehörigen B^+-Bäume zurückgeliefert. Der Node-Identification-Algorithmus ermittelt aus den Wurzeln der B^+-Bäume und den Suchschlüsseln bei erfolgreicher Suche die Pointer auf den Dokument-Graphen. Suchschlüssel können je nach Anfrage exakte Matches sein oder Prädikate, die einen Bereich bestimmen. Als letzter Schritt wird der Dokument-Graph ausgehend von den Pointern auf die relevanten Knoten traversiert und die Ergebnisse ausgegeben. Ein Beispiel für eine konkrete Anfrageverarbeitung wird in Kapitel 3.1 erläutert.

3 Ein Vergleich - SphinX vs. Lore

3.1 Anfrageverarbeitung

Während Sphinx nur einen einzelnen Index benutzt, kommt bei Lore neben den bereits vorgestellten Indizes Tindex, Vindex und Pindex noch der Lindex zum Einsatz. Dieser Index dient zur Identifizierung der Elternknoten, die über ein gegebenes Label von einem Knoten aus erreichbar sind. Die unterschiedliche Struktur der beiden Indizes bewirkt einen großen Unterschied bei der Anfrageverarbeitung. Dies wird im folgenden Beispiel verdeutlicht.
Beispielanfrage zu Abbildung 2.11
A1: Suche /bib[//year=2001]//conference
(Suche die Namen aller Konferenzen, die im Jahr 2001 stattfanden und zu denen ein Eintrag in der Datenbank existiert.)
A1': Suche /bib/article[year=2001]/conference
(Suche die Namen aller Konferenzen, die im Jahr 2001 stattfanden und zu denen ein Artikel in der Datenbank existiert.)
Es ist zu beachten, dass Lore als Datenmodell einen kantenbeschrifteten Graph verwendet, während SphinX einen knotenbeschrifteten Graph einsetzt. Ein eingehendes Label bei Lore entspricht also dem Typ des Elternknotens bei SphinX. Abbildung 2.11 zeigt das Datenmodell von SphinX.
Bei SphinX werden zuerst mit Hilfe des Index-Path-Extraction-Algorithmus die möglichen absoluten Pfade aus den allgemeinen Pfaden extrahiert. Dies sind bei Anfrage A1 /bib/book/year und /bib/article/year für /bib//year und /bib/article/conference für /bib//conference. Ein Join der resultierenden Pfade ergibt die relevanten absoluten Pfade /bib/article/year für /bib//year und /bib/article/conference für /bib//conference. Der B^+-Baum über /bib/article/year liefert das Ergebnis &23 für „2001". Im letzten Schritt wird nun der Dokument-Graph ausgehend von &23 entsprechend der relevanten einfachen Pfade traversiert und liefert „ICDT". Bei Anfrage A1' werden durch den Index-Path-Extraction-Algorithmus direkt die relevanten absoluten Pfade (also /bib/article/year und /bib/article/conference) ausgegeben, ansonsten läuft die Verarbeitung analog.
Lore benutzt einen über „year" gebildeten Vindex um alle Vorkommnisse von „2001" zu extrahieren. Da der Vindex nur das eingehende Label (in Abbildung 2.11 also den Typ des Elternknotens) zu einem Blatt speichert, untersucht er für Anfrage A1 Vorkommnisse von „year" für /bib/article/year und /bib/book/year. Als Ergebnis wird also &23 und &11 geliefert. Nun werden mit dem Lindex die Elternknoten extrahiert, die über das Label „year" erreichbar sind. Für &23 ist das &17 (über das Label „article") und für &11 ist es &10 (über das Label „book").Von diesen Knoten ausgehend sucht nun der Pindex alle Knoten, die über //conference erreichbar sind. Da &10 ein unproduktives Ergebnis war, wird nur „ICDT" für &17 zurückgeliefert. Für Anfrage A1' liefert der Vindex das gleiche Ergebnis, da nur die eingehenden Label bekannt sind. Erst der Lindex erkennt den unproduktiven Pfad und liefert nur noch &17. Der Rest verläuft analog zu A1.
Der Leistungsunterschied zwischen Lore und SphinX wird deutlich, wenn man den Fall betrachtet, dass eine Datenbank z. B. tausend Einträge für /bib/book[year=2001] enthält, aber nur einen mit /bib/article[year=2001]. Während SphinX sowohl für A1 als auch für A1' nur den einen produktiven Eintrag untersucht, betrachtet Lore in beiden Anfragen auch alle tausend unproduktiven Pfade.

3.2 Experimenteller Aufbau

Der Leistungsvergleich wurde auf einem Intel Pentium III 800 MHz mit 512 MB Speicher und 18 GB SCSI Plattenspeicher durchgeführt. Als Betriebssystem kam Redhat Linux 7.1 zum Einsatz. Sowohl die Daten als auch der Index wurden auf Platte gespeichert.

Verwendete Dokumente

- Shakespeare : Öffentlich erhältliche XML Version der Stücke von Shakespeare, die viele lange Textpassagen enthalten [Bos99].
- Conference, Journal: Repräsentieren die Conference- und Journal-Einträge des DBLP-Archives [Ley03]
- Swiss-Prot: Protein-Sequenz Datenbank
- HAM-RADIO: Dokumente aus der FCC-Ham-Radio-Datenbank [ICT01] mit einem hohen Anteil an Metadaten
- Xmark: Synthetisches Dokument, generiert von xmlgen, dem Daten-Generator von Xmark [SWK01]. Das Dokument stellt eine Auktions-Webseite dar.

Tabelle 6 zeigt die Statistiken zu den verschiedenen Dokumenten. Records ist die Zahl der Top-Level-Einträge, Depth die maximale Tiefe der Verschachtelung, Elems und Attrs die Anzahl der Elemente bzw. Attribute in der DTD und Total Elems und Total Attrs die Anzahl der Vorkommnisse von Elementen und Attributen im Text.

Document	Type	Size (in MB)	Records	Depth	Elems	Attrs	Total Elems	Total Attrs
Shakespeare	Real	7.4	35	High	22	0	179619	0
CONFERENCE	Real	40	104K	Low	25	2	1029494	159928
JOURNAL	Real	27	76K	Low	15	2	804176	89567
SWISS-PROT	Real	158	80k	High	-	-	4243031	2898833
HAM-RADIO	Real	361	0.70M	Low	24	0	14117198	0
XMARK	Synthetic	1126	1	High	77	16	16703249	3829772

Tabelle 6: Dokument- Statistiken

Anfragen

Es wurde die folgende Menge von Anfragen verwendet, unterteilt in QS für einfache Pfadausdrückes, also voll spezifizierte Pfade, und QG für allgemeine Pfadausdrücke. Anfrage 1 ist für SHAKESPEARE, Anfragen 2 und 3 für CONFERENCE, Anfragen 4 und 5 für JOURNAL, Anfrage 6 für HAM-RADIO und Anfrage 7 für Xmark. Die Anfragen sind im Xpath-Format angegeben.

QS1: /Shakespeare/Play[Title=“The Tragedy of Antony and Cleopatra”]/Personae/Persona
QG1: //[Title=“The Tragedy of Antony and Cleopatra”]//Personae/Persona

QS2: /conference/proceedings[year=”1998”]/title
QG2: //proceedings[year=”1998”]/title

QS3: /conference/inproceedings[year=”1995”]/booktitle
QG3: //[year=”1995”]//booktitle

QS4: /journal/article[title=”Introduction”]/author
QG4: //[title=”Introduction”]//author

QS5: /journal/article[journal=”Algorithmica”]/author
QG5: //[journal=”Algorithmica”]//author

QS6 : /FMDataBase/FCCAmRadio[Adress.city=”BROOKLYN”]/LicenceNo
QG6: //[Adress.city=”BROOKLYN”]//LicenceNo

QS7: /site/open_auctions/open_auction[type=”Regular”]/Privacy
QG7: //[type=”Regular”]//Privacy

3.3 Ergebnisse

Index-Konstruktion
Die Index-Größe und -Erstellungszeit sind in den Tabellen 7 und 8 dargestellt. In beiden Fällen zeigt sich, dass Lore einen 2- bis 4-fach höheren Platzbedarf hat und die 3- bis 33-fache Zeit benötigt. Bei den beiden größten Dokumenten kommt es bei Lore sogar zu einem vorzeitigen Abbruch. Im Gegensatz dazu erreicht SphinX bei 1.1 GB Daten sogar einen Durchsatz von fast 1 MB pro Sekunde.

Document	SphinX	Lore
Shakespeare	15M	35M
CONFERENCE	85M	226M
JOURNAL	63M	230M
SWISS-PROT	465M	877M
HAM-RADIO	834M	-Incomplete-
Xmark	1900M	-Incomplete-

Tabelle 7: Index- Größe

Document	SphinX	Lore
Shakespeare	14sec	41sec
CONFERENCE	94sec	1433sec
JOURNAL	73sec	969sec
SWISS-PROT	614sec	20460sec
HAM-RADIO	1220sec	- Incomplete -
Xmark	1930sec	- Incomplete -

Tabelle 8: Erstellungszeit

Leistung bei der Anfrageverarbeitung
Die Antwortzeiten sind in den Tabellen 9 und 10 aufgelistet. Auch hier war Lore wieder schlechter, in manchen Fällen kam es zu einem vorzeitigen Abbruch. Die Lücke zwischen Lore und SphinX wächst, je größer das Dokument ist. Der Grund für den geringen Unterschied zwischen einfachen und allgemeinen Pfadausdrücken bei Lore liegt darin, dass die Anzahl der zusätzlichen unproduktiven Pfade durch allgemeine Pfadausdrücke gegenüber der Gesamtzahl der unproduktiven Pfade gering ist (siehe Kapitel 3.1).

Document	Query#	SphinX	Lore
Shakespeare	QS1	0.7s	3.3s
CONFERENCE	QS2	0.5s	-Incomplete-
CONFERENCE	QS3	6.8s	-Incomplete-
JOURNAL	QS4	0.8s	59.9s
JOURNAL	QS5	0.7s	60.7s
HAM-RADIO	QS6	122.8s	-Incomplete-
Xmark	QS7	234.2s	-Incomplete-

Tabelle 9: Zeitbedarf für einfache Pfadausdrücke

Document	Query#	SphinX	Lore
Shakespeare	QG1	0.8s	5.4s
CONFERENCE	QG2	1.1s	-Incomplete-
CONFERENCE	QG3	7.8s	-Incomplete-
JOURNAL	QG4	0.8s	61.2s
JOURNAL	QG5	0.7s	62.6s
HAM-RADIO	QG6	140.8s	-Incomplete-
Xmark	QG7	237.2s	-Incomplete-

Tabelle 10: Zeitbedarf für allgemeine Pfadausdrücke

4 Schlusswort

Der Bereich Indexstrukturen für teilweise strukturierte Daten bzw. XML wird zur Zeit sehr intensiv erforscht. Es gibt eine große Menge von aktuellen Publikationen, die allerdings, wie schon von Zobel, Moffat und Ramamohanarao im Jahr 1996 (!) in [ZMR96] bemängelt, die Strukturen sehr unvollständig beschreiben und untersuchen. Da die Leistung der vorgestellten Indexstrukturen häufig von den zu indexierenden Daten abhängig ist, wäre ein direkter Vergleich der Indexstrukturen von großer Hilfe für die Bewertung. In den Publikationen werden meist nur Vergleiche gegenüber einem System ohne Indizes vorgestellt. Da aber jeder Autor mit anderen Daten arbeitet, ist ein Vergleich sehr schwierig. Auch werden die Nachteile einer Struktur häufig verschwiegen.
In der vorliegenden Arbeit wurden die Indexstrukturen in drei Klassen eingeteilt: Inhaltsbasierte Indizes, strukturbasierte Indizes und hybride Indizes. In der Literatur sind allerdings auch andere Klassifikationen zu finden (z. B. [Wei02]). Für jede der Klassen wurden repräsentativ zwei Indexstrukturen vorgestellt. Dies waren für inhaltsbasierte Indizes Text Index und Value Index, für strukturbasierte Indizes DataGuide und T-Index und für hybride Indizes IndexFabric und BUS. Außerdem wurde noch SphinX vorgestellt. Dieser Index gehört eigentlich zu den hybriden Indizes, wurde aber extra aufgeführt, da hier die vorliegende Struktur der Daten in Form der DTD zur Indexierung genutzt wird.
Zum Abschluss wurde noch ein Leistungsvergleich zwischen SphinX und Lore dargestellt, welches die einzigen Systeme sind, für die ein direkter Vergleich existiert.

5 Literaturnachweis

[Bos99] J. Bosak. The Plays of Shakespeare in XML, 1999.
WWW: http://www.oasis-open.org/cover/bosakShakespeare200.html

[CSF+01] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases* (VLDB), Roma, Italy, pp. 341-350, 2001.
WWW: http://citeseer.nj.nec.com/cooper01fast.html

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Technical Report, Stanford University, Computer Science Department, Database Group, 1997.
WWW: http://dbpubs.stanford.edu:8090/pub/1997-50

[GBYS91] G.Gonnet, R. Baeza-Yates and T. Snider. Lexicographical Indices for Text: Inverted Files vs. PAT trees. Technical Report OED-91-01, Centre for the New OED, University of Waterloo, February 1991.
WWW: http://citeseer.nj.nec.com/gonnet91lexicographical.html

[ICT01] Intelligent Compression Technologies ICT , XML Testfiles
FTP: ftp://ftp.ictcompress.com/pub/xmltestfiles

[Ley03] M. Ley. Computer Science Bibliography, Trier, 2003.
WWW: http://www.informatik.uni-trier.de/~ley/db/

[MS99] Tova Milo and Dan Suciu. Index Structures for Path expressions. In *Proceedings of the 7th International Conference on Database Theory* (ICDT), Jerusalem, Israel, pp. 277-295, January1999.
WWW: http://citeseer.nj.nec.com/milo97index.html

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo and A. Rajamaran. Indexing Semistructured Data. Technical Report, Stanford University, Computer Science Department, Database Group, January 1998.
WWW: http://citeseer.nj.nec.com/mchugh98indexing .html

[OSS01] V. Oria, A. Shah and S. Sowell. Indexing XML Documents: Improving the BUS Method. In *MIS '01, 7th Workshop on Multimedia Information Systems, 7-9 November 2001, Capri, pp. 51-60.*

[PH01] Leela Krishna Poola and Jayant R. Haritsa. SphinX: Schema-conscious XML Indexing. Database Systems Laboratory, Dept. of Computer Science & Automation, Indian Institute of Science, Bangalore, India, 2001
WWW: http://citeseer.nj.nec.com/poola01sphinx.html

[SWK01] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. J. Carey and R. Busse. The XML Benchmark Project, April 2001.
WWW: http://monetdb.cwi.nl/xml/Benchmark/benchmark.html

[Wei02] Felix Weigel. A Survey of Indexing Techniques for Semistructured Documents. Project Thesis, 2002.
WWW: http://www.pms.informatik.uni- muenchen.de/publikationen/projektarbeiten/Felix.Weigel/master.pdf

[Wid99] J. Widom. Data management for XML: Research directions. *IEEE Data Engineering Bulletin, 22(3): pp. 44-52, 1999.*
WWW: http://citeseer.nj.nec.com/widom99data.html

[ZMR96] J. Zobel, A. Moffat and K. Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. In *ACM SIGMOD Record 25, 3 (Oct), pp. 10-15, 1996.*

[ZMR98] J. Zobel, A. Moffat and K. Ramamohanarao. Inverted Files Versus Signature Files for Text Indexing. In *ACM Transactions on Database Systems 23(4) (1998) pp. 453-490.*
WWW: http://citeseer.nj.nec.com/88449.html