XML-Indexierung

Fabian Fichter

Technische Universität Kaiserslautern, Kaiserslautern, Germany
f_fichte@informatik.uni-kl.de

Zusammenfassung. Die *extensible markup language (XML)* spielt eine immer größer werdende Rolle beim Austausch und Speichern von Daten. Da es sich oft um große Datenmengen handelt, müssen diese indexiert werden, um einen effizienten Zugriff zu ermöglichen. Während dies in klassischen Datenbank- und Information-Retrieval-Systemen schon lange zum Standard gehört und sehr gut erforscht ist, ist die Indexierung von XML-Dokumenten ein aktuelles Forschungsgebiet. Inhalt dieser Arbeit ist eine grundlegende Einführung in die Indexierung von XML-Dokumenten im Hinblick auf den XML-spezifischen Aufbau und der zugrunde liegenden Datenstrukturen.

1 Einleitung

Nur wenige Jahre nach der Einführung des XML-Standards[1] (1998) wurde der Begriff *XML* zu einem der meistgenutzten Schlagwörter in der Informationstechnologie. Dies hat mehrere Gründe. Erstens ist der XML-Standard für jeden frei verfügbar und einfach zu verstehen. XML-Dokumente können leicht erstellt und ausgewertet werden und sind für Menschen lesbar und nachvollziehbar. Zweitens ist XML eine Auszeichnungssprache zur Darstellung hierarchisch organisierter Daten, und eignet sich somit als Basis für eine Vielzahl von Datenformaten. Der dritte Grund für die Popularität, und auch als Auslöser zu verstehen, ist die immer größer werdende Vernetzung von Rechnersystemen und die massive Nutzung des Internets. Der Bedarf, Informationen auszutauschen, wuchs rapide an, und hierfür waren Austauschformate notwendig, die gerade vorige Eigenschaften erfüllten. Es sind auch andere Auszeichnungssprachen möglich (SGML, HTML, proprietäre Formate), jedoch ist, neben oben genannten Vorteilen, bei XML das Verhältnis zwischen der Komplexität und Erweiterbarkeit für eine Vielzahl von Anwendungsgebieten geeignet. Der Standard ist spezifisch genug, um die Modellierungskomplexität zu reduzieren, aber offen genug, um gut an individuelle Bedürfnisse angepasst zu werden.

Ein XML-Dokument besteht aus einer Menge von Elementen, die hierarchisch strukturiert sind. Diese Elemente bestehen aus einem Start- und einem End-Tag zwischen denen der Inhalt des Elements steht, welcher aus weiteren Elementen und purem Text bestehen kann. Tags sind Bezeichner, die den Inhalt eines Elements beschreiben; neben einer strukturellen haben sie somit auch eine semantische Bedeu-

[1] Genau genommen ist XML kein Standard, sondern eine Empfehlung des World Wide Web Consortiums (W3C). Die Empfehlungen des W3C haben jedoch den Charakter von Standards.

tung. Weitere Daten können in den Attributen eines Elements gespeichert werden. Dabei sind verschiedene Attribute (z. B. IDREF) zum Referenzieren von anderen Elementen festgelegt. Durch diesen Aufbau können XML-Dokumente als Baum, bzw. durch die Referenzen als allgemeiner Graph modelliert werden. Aus dieser spezifischen Eigenschaft folgt, dass im Gegensatz zu traditionellen relationalen Datenbanken und Information-Retrieval-Systemen, hier auch Pfadanfragen unterstützt werden müssen. Anfragen werden typischerweise auf Grundlage der *XPath* Spezifikation modelliert. Dabei werden Unterelemente durch einen Slash gekennzeichnet und weitere Bedingungen auf diesen durch eckige Klammern eingeleitet. Folgende Anfrage greift z. B. auf das erste Element von inventarliste über haus und dann zimmer zu:

```
haus/zimmer/inventarliste/first.
```

Um Anfragen effizient umzusetzen, ist eine Indexierung der Dokumente notwendig. Aufgrund des Aufbaus eines XML-Dokuments können Anfragen Bezug auf den Inhalt (reiner Text), auf die Werte (Elementnamen, Attributnamen, Attributwerte) und auf die Graphenstruktur nehmen. Dies führt zu drei grundlegenden Indexierungsarten: *Volltextindeex, Werteindexe* und *Strukturindexe*.
Nachfolgend werden allgemeine Grundlagen zur Indexierung, und dann darauf aufbauend die drei Indexierungsarten erläutert. Der Focus liegt dabei auf der Strukturindexierung, da diese einen grundlegenden Unterschied zu relationalen Datenbanken darstellt.

2 Grundlagen der Indexierung

Unter Indexierung versteht man das Erstellen einer Zugriffsstruktur (Indexstruktur), deren Elemente mit den eigentlichen Daten verknüpft sind. Ein Index besteht dann aus einer Datenstruktur, die auf die Daten verweist, und Algorithmen, die über die Indexstruktur bestimmte Operationen umsetzen. Allgemein gehören dazu: *Suche (search)*, *sequentiell durchlaufen (scan)*, *einfügen (insert)*, *löschen (delete)* und *ändern (update)*. Aufgrund der Ähnlichkeit der Operationen können *search/update* und *insert/delete* zu Gruppen zusammengefasst werden. Oft liegt der Schwerpunkt auf einer dieser Gruppen, was in der Regel zu einer Leistungsteigerung dieses Operationstyps auf Kosten des anderen führt.

Indexe kommen somit überall dort vor, wo effizient auf großen Datenmengen gearbeitet werden muss, wie z. B. bei Dateisysstemen (HFS, ext2, NTFS), beim Information Retrieval (Suchmaschinen) und allgemein bei Datenbanksystemen (db2, postgeSQL, etc.).

Schwerpunkt dieses Kapitels sind grundlegende Indexstrukturen, auf welchen die Indexe in Kapitel 3 aufbauen.

2.1 Eigenschaften von Indexen

Basis für einen Index sind Schlüssel, welche die zu indexierenden Objekte eindeutig identifizieren. Für jedes Objekt wird der Schlüssel aus bestimmten Eigenschaften (z. B. Elementnamen bei einem XML-Dokument oder die ganze Zeichenkette bei Wör-

tern) generiert, und in die Indexstruktur, zusammen mit einem Verweis auf die Daten, eingefügt.

Indexe können durch die Art der Indexstruktur klassifiziert werden: Suchbäume, Hashtabellen und Bitmaps. Hiervon sind vor allem die ersten beiden für die XML-Indexierung interessant und werden in den folgenden Abschnitten erläutert.

2.2 Auswahlkriterien

Um für eine bestimmte Datenbank die richtige Indexstruktur zu finden, sind eine Reihe von Entscheidungskriterien notwendig. Das wichtigste Kriterium bei der Auswahl einer Indexstruktur ist deren Anwendbarkeit: Welche Typen von Anfragen werden von der Indexstruktur unterstützt? Jede Indexstruktur unterstützt nur eine Teilmenge aller möglichen Anfragen. Es gibt auch Klassen von Anfragen, die nur durch eine komplette Suche auf der Datenbank beantwortet werden können. Von Interesse ist auch die Erweiterbarkeit einer Indexstruktur auf andere Datentypen. Da in den letzten Jahren die Speicherkapazitäten von Datenträgern stark zugenommen haben, ist auch die Skalierbarkeit ein wichtiger Faktor. Kann die Indexstruktur auch die 1000-fache Datenmenge effizient verwalten? Eine wichtige Rolle spielen außerdem die Auswertungszeit und der Speicherplatzbedarf. Ein weiterer Aspekt ist die Update-Fähigkeit der Indexstrukturen. Falls sich die indexierten Datenmengen häufig ändern, ist dies ein wichtiges Kriterium. Bei einigen Indexstrukturen müssen nur kleine Änderungen vorgenommen, andere komplett neu erstellt werden.

2.3 Suchbäume

Grundlage der folgenden Indexe ist der binäre Suchbaum. In der Regel bieten solche Indexe einen Zugriff in O(log n), und Bereichsanfragen sind möglich. Abbildung 1 zeigt einen ersten Index, auf Basis eines binären Suchbaums, für die Menge K = {4, 5, 8, 9, 12, 13, 14, 16, 18, 19, 33, 44, 45}

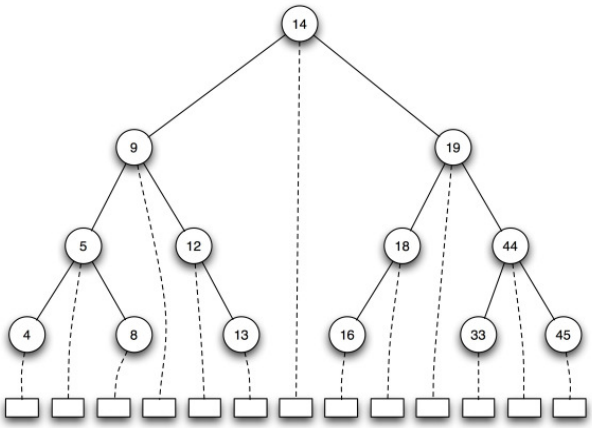

Abb. 1. einfacher Index für K = {4, 5, 8, 9, 12, 13, 14, 16, 18, 19, 33, 44, 45}

In den Knoten ist jeweils noch ein Verweis auf das indexierte Objekt gespeichert. Um ein Objekt zu finden, muss man jeden Knotenschlüssel mit dem des Objektes vergleichen. Ist der Objektschlüssel größer, steigt man rechts ab, falls er kleiner ist links. Dies wird solange fortgeführt, bis der Schlüssel gefunden ist, oder man nicht weiter absteigen kann.
Durch einen derartigen Aufbau erhält man einen ersten, einfachen Index. Bei sehr großen Datenmengen ist dieses Verfahren jedoch nicht mehr effizient, da die Indexstruktur dann nicht mehr in den Hauptspeicher passt und somit sehr häufig (im schlimmsten Fall für jede Verzweigung) auf das externe Speichermedium zugegriffen werden muss. Um dies zu lösen, speichert man mehrere Schlüssel in einer Seite und kann so durch einen Zugriff auf das Speichermedium viele Schlüssel erhalten. Dies ist die Grundidee von den nun folgenden B- und B*-Bäumen.

2.3.1 B-Baum, B*-Baum

Mit dem B-Baum, und seiner Weiterentwicklung B*-Baum, lassen sich effiziente Indexstrukturen für große Datenmengen erstellen, die nicht in den Hauptspeicher passen. Während der B-Baum mehr eine Speicherstruktur ist, stellen der B*-Baum und Derivate echte Indexstrukturen dar, deren Grundlage breite, niedrige Bäume sind. In den Knoten werden ganze Seiten von geordneten Schlüsseln gespeichert, um die Lokalität der Daten auf der Festplatte auszunutzen. Dadurch erhält man pro Seitenzugriff mehrere geordnete Schlüssel, die man zum Vergleichen nutzen kann, was die Anzahl der externen Speicherzugriffe je nach Seitengröße drastisch reduziert.

B-Baum

Beim B-Baum handelt es sich in erster Linie um eine Speicherstruktur, da er aber Grundlage für den B*-Baum ist, sei er an dieser Stelle kurz erläutert.

Definition[2]
Seien k, h ganze Zahlen, $h \geq 0$, $k > 0$. Ein B-Baum der Klasse $\tau(k, h)$ ist entweder ein leerer Baum oder ein Suchbaum mit folgenden Eigenschaften:

- Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge h-1.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne. Die Wurzel ist entweder ein Blatt oder hat mindestens 2 Söhne.
- Jeder Knoten hat höchstens 2k+1 Söhne.
- Jedes Blatt mit Ausnahme der Wurzel als Blatt hat mindestens k und höchstens 2k Einträge.

[2] Siehe [HÄR]

Somit ergibt sich folgendes Knotenformat:

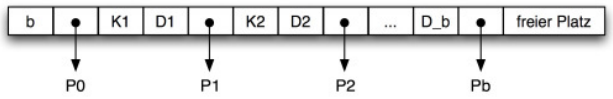

Abb. 2. Knotenformat eines B-Baums

Folgende Abbildung zeigt einen B-Baum. Die Knoten werden nur durch die Schlüssel angedeutet, haben aber obiges Format.

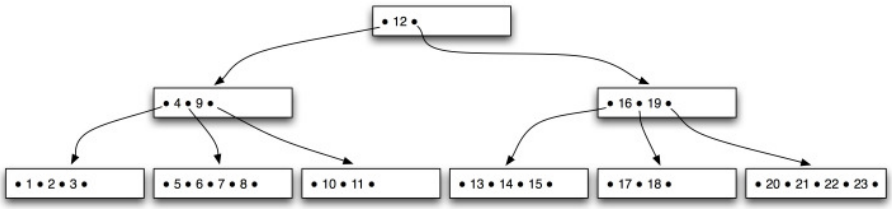

Abb. 3. Ein B-Baum der Klasse $\tau(2,3)$

B*-Baum

Im Unterschied zum B-Baum werden hier die Daten nur in den Blättern gespeichert. In den Knoten kommen nur die Schlüssel vor, und in den Blättern stehen Paare von Schlüsseln und Daten. Somit haben die Knoten eine wegweisende Funktion und in den Blättern stehen die eigentlichen Daten. Die inneren Knoten repräsentieren damit einen Index, über den man schnell auf die eigentlichen Daten zugreifen kann. Man kann somit leicht die Daten auf einem Sekundärspeicher auslagern und trotzdem effizient und ohne unnötige Zugriffe auf den Sekundärspeicher darauf zugreifen. Zwar ergibt sich für einige Schlüssel eine redundante Speicherung (einmal im Index und dann noch einmal in den Blättern) – dies ist bei größeren Datenbeständen jedoch nicht weiter relevant.

Definition[3]

Seien k, k* und h* ganze Zahlen, $h^* \geq 0$, $k, k^* > 0$. Ein B*-Baum der Klasse $\tau(k, k^*, h^*)$ ist entweder ein leerer Baum oder ein Suchbaum, für den gilt:

- Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge h*-1.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.
- Jeder innere Knoten hat höchstens 2k+1 Söhne.
- Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k* und höchstens 2k* Einträge.

[3] Siehe [HÄR]

Anschaulich kann man den Indexteil als B-Baum verstehen, der auf der untersten Ebene auf eine sequentielle Anordnung der Daten-Schlüssel-Paare verweist. Abbildung 4 zeigt einen B*-Baum der Klasse $\tau(2,2,2)$

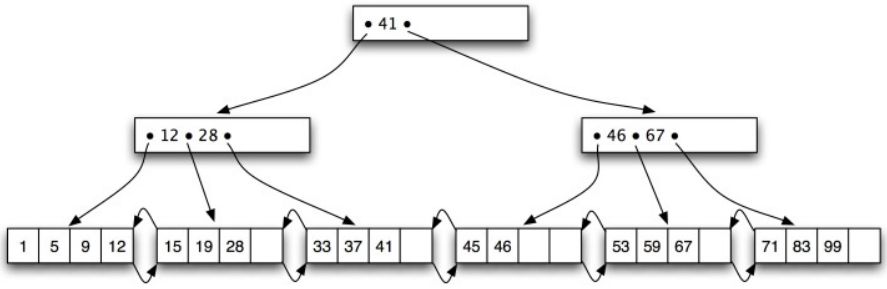

Abb. 4. B*-Baum der Klasse $\tau(2,2,2)$

Suche
Entsprechend den Schlüsseln in einer Seite wird dem passenden Verweis gefolgt, bis man bei den Blättern angekommen ist. Dies führt zu h* Seitenzugriffen.

Sequentielle Suche
Als Erstes wird der äußerste, linke Teil gesucht. Von diesem Blatt ausgehend kann man nun die sequentielle Verkettung der Blätter ausnutzen.

Einfügen
Das Einfügen ist sehr ähnlich zum B-Baum. Ist eine Seite voll, wird sie gesplittet, was durch eine Überlaufbehandlung durchgeführt wird.

Überlaufbehandlung
Durch den Überlauf eines Blattknotens muss der Baum teilweise reorganisiert werden. Zuerst wird der Blattknoten in zwei neue aufgeteilt (gesplittet). Die k*/2+1 ersten Elemente werden im ersten Knoten gespeichert und die nächsten k*/2 werden im zweiten Knoten abgelegt. Der neue Blattknoten muss im Vaterknoten vermerkt werden. Dadurch kann auch dieser überlaufen. Für diesen Knoten werden dann zwei neue angelegt, auf welche die k/2 ersten und k/2 letzten Verweise verteilt werden. Der mittlere Eintrag wird nach oben weitergegeben. Dies kann sich so lange fortsetzten, bis eine neue Wurzel angelegt werden muss, und die Höhe des Baumes um eins größer wird.

Löschen
Es wird nur in den Blättern gelöscht. Die zurückbleibenden Schlüssel dienen weiterhin als Wegweiser. Sind zu wenig Einträge im Blatt, folgt eine Unterlaufbehandlung.

Unterlaufbehandlung
Beim Löschen kann es passieren, dass weniger als k*/2 Einträge in dem betreffenden Blatt gespeichert sind (Bedingung 4 von der B*-Baum Definition). Man spricht dann von einem *Unterlauf*. Zu Beginn wird geprüft, ob man durch Umverteilung der Einträge eines benachbarten Blattes wieder mindestens k*/2 Einträge in beiden Blättern erreichen kann. Ist dies nicht möglich, passen alle Einträge in ein einzelnes Blatt, und die beiden ursprünglichen werden verschmolzen. Der Prozess ist analog zur Überlaufbehandlung und kann sich bis zur Wurzel fortsetzen, was die Höhe des Baumes um eins verringern kann.

Splitfaktor
Bei der Unterlaufbehandlung wird zuerst geprüft, ob man Einträge eines Nachbarblatts umverteilen kann. Dies ist auch beim Einfügen nützlich und führt zu besser gefüllten Knoten.

Der Splitfaktor beschreibt, bei wie vielen Nachbarknoten geprüft wird, ob eine Umverteilung stattfinden kann. Bei einem Splitfaktor von eins wird nur der aktuelle Knoten betrachtet. Bei einem Faktor von zwei auch der rechte Nachbar, dann der linke usw. Durch die bessere Speicherausnutzung wächst der Baum nicht so schnell und die Suche verläuft somit schneller. Der Nachteil ist der höherere Aufwand beim Einfügen.

2.4 Hashtabellen

Basis der verschieden Varianten sind Hashfunktionen, welche die Schlüssel auf die Speicherplätze möglichst verteilt abbilden. Dadurch erhält man theoretisch einen Zugriff auf ein Element in O(1). Die Speicherplätze werden bei Hashverfahren *Buckets* genannt, und entsprechen Seiten. Bei der Abbildung der Schlüssel durch die Hashfunktion kommt es zwangsläufig vor, dass zwei verschiedene Schlüssel den gleichen Bucket zugewiesen bekommen. Dies wird Schlüsselkollision bezeichnet. Der Nachteil von Hashtabellen ist, dass Bereichsanfragen nur sehr ineffizient umgesetzt werden können[4].
Hashverfahren können in *statisches* und *dynamisches* Hashing unterteilt werden, wobei dynamische Verfahren wegen der besseren Skalierbarkeit als Datenbankindex interessant sind. Aus diesem Grund wird im Folgenden nur ein Vertreter des dynamischen Hashings, das *erweiterbare* Hashing, vorgestellt werden.

Erweiterbares Hashing
Das erweiterbare Hashing besteht aus einem Verzeichnis, in welchem Verweise auf die einzelnen Buckets gespeichert werden. Das Verzeichnis ist ein Vektor mit 2d Einträgen, die jeweils eine Bucket-Adresse speichern. Die aktuelle Verzeichnisgröße wird als globale Verzeichnistiefe d bezeichnet und die Schlüssel werden auf die einzelnen Verzeichniseinträge durch eine Hashfunktion abgebildet. Dabei dienen die letzten d Bit des Rückgabewerts als Index im Verzeichnisvektor für den Adresseintrag des zugehörigen Buckets. Desweiteren können mehrere Verzeichniseinträge auf das

[4] In der Regel muss ein Scan über die ganzen Daten durchgeführt werden.

gleiche Bucket verweisen, wenn die letzten d' Bit des Verzeichnisindexes gleich sind. Diese Mehrverweise werden in den Buckets als *lokale Buckettiefe* bezeichnet und abgespeichert. Somit ergeben sich jeweils 2d-d' Verzeichniseinträge, die auf einen gemeinsamen Bucket verweisen.

2.5 Zusammenfassung

Mit Bäumen lassen sich effizient Bereichsanfragen und Sortierungen unterstützen, wohingegen dies durch Hashtabellen nicht gut möglich ist. Dafür können Punktanfragen sehr schnell und mit konstantem Aufwand ausgeführt werden.

3 Verfahren zur XML-Indexierung

Im Gegensatz zum klassischen Information Retrieval besitzen XML-Dokumente auch Strukturinformationen in Form von Tags und Attributen. Da diese Struktur jedoch keinem festen Schema folgt, lässt sich XML-Indexierung nicht gleichsetzen mit objektorientierten Ansätzen oder relationalen Datenbanken. Demnach fließen in die Indexierung von XML-Dokumenten Ansätze und Techniken aus diesen Welten plus XML-spezifischer Verfahren ein.
Grundlegend kann man drei Typen unterscheiden: *Werteindexe, Volltextindexe* und *Strukturindexe*. Während die ersten beiden mit klassischen Verfahren zur Indexierung umgesetzt werden können, ist der letzte Typ XML-spezifisch. Hierbei werden Anfragen, die die Graphenstruktur eines XML-Dokuments betreffen, ermöglicht. Bezüglich der Strukturindexierung werden in dieser Arbeit die Pfadindexierung und die baumbasierte Indexierung behandelt.
Um Anfragen die den Inhalt und die Struktur umsetzen zu können, braucht man eine Kombination aus obigen Verfahren. Dies führt zu *hybriden Indexen*.

In den folgenden Abschnitten werden diese Verfahren, und Implementierungen davon, näher erläutert. Der Fokus liegt dabei auf den Pfadindexen.

3.1 Graphenstruktur eines XML-Dokuments

Verfahren, die die Struktur eines XML-Dokuments berücksichtigen, basieren immer auf der Möglichkeit, ein XML-Dokument als markierten, gerichteten Graphen zu sehen. Im Speziellen kann dies auch ein Baum sein, jedoch können Referenzen in den Attributen zu einem Graphen führen. Abbildung 6 zeigt den Graphen für folgendes XML-Dokument.

```
<bib>
  <book RefID="1997">
    <title>Elements of ML programming</title>
    <author> <lastname>Ullman </lastname> </author>
    <publisher> <name>Prentice Hall </name>
    <address>New Jersey 07458<address> </publisher>
  </book>
  <book RefID="2001">
    <title>Foundation for Object/Relational Databases</title>
    <author><lastname>Darwen </lastname> </author>
    <publisher><name> Addison−Wesley </name></publisher>
  </book>
  <article ID="2001">
    <author> <lastname>Ullman</lastname>
          <firstname> Jeffrey D.</firstname>
    </author>
    <title> Querying Websites Using Compact Skeleton </title>
    <conference> ICDT </conference>
  </article>
</bib>
```

Abb. 5. Beispiel XML-Dokument

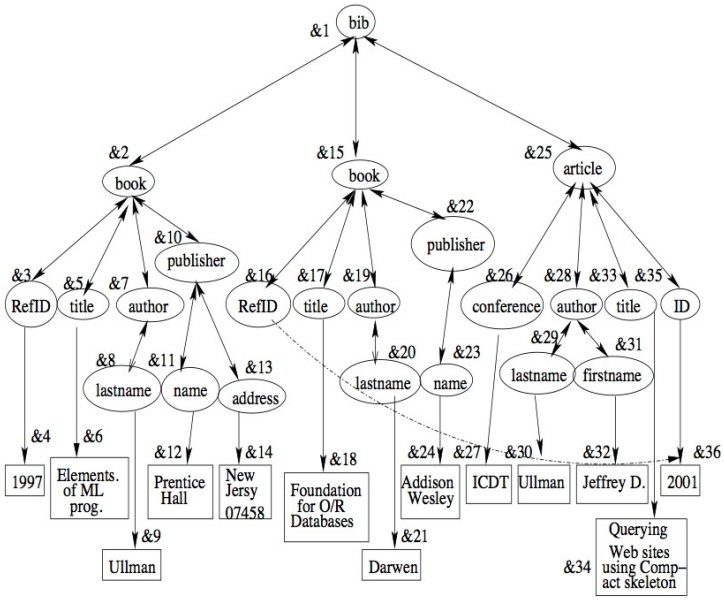

Abb. 6. Beispielgraph

3.2 Werteindex

Im Datenbankbereich sind Werteindizes als sogenannte *Zugriffspfade* bekannt und dienen zum schnellen Zugriff auf Datensätze über Werte oder Wertkombinationen von z.B. Attributen einer Relation. Im Fall von XML soll der Zugriff auf atomare Werte wie Elementinhalte, numerische Werte oder Attributwerte spezieller Elemente unterstützt werden, und ist für strukturierte Anteile eines XML-Dokuments interessant. Auch kann man auch schnell auf Elementnamen oder KnotenIDs zugreifen.

Geeignete Indexstrukturen hierfür sind[5]:

- dynamische Indexbäume wie der B*-Baum
- Hashverfahren wie das erweiterbare Hashing
- Signaturbäume.

3.3 Volltextindex

Die Volltextindexierung kommt aus dem Information Retrieval und befasst sich mit der Indexierung von Worten der natürlichen Sprache. Sie kommt überall vor, wo große Sammlungen von Texten natürlicher Sprache indiziert werden müssen.

Vor der eigentlichen Indexierung findet in der Regel eine Normalisierung des Volltextes durch eine oder mehrere der folgenden Methoden statt:

- Erkennung von Interpunktionszeichen, um logische Texteinheiten für eine Phrasensuche zu erhalten.
- Eliminierung von häufig auftretenden Worten mit geringer Bedeutung; dies geschieht über eine Stoppwortliste, in welcher die nicht zu indexierenden Worte aufgelistet sind.
- Durchführung einer Linguistische Normalisierung oder Grundformbildung (Substantiv, Nominativ, Singular, Verben: Infinitiv).

3.3.1 Anfragetypen von Volltextindexen

Die unterstützten Anfragetypen zählen zu den wichtigen Eigenschaften zur Differenzierung von Volltextindizes. Sie sollen im Nachfolgenden erläutert weren.

Stichwortsuche und boolsches Retrieval
Basierend auf den ganzen Worten einer Suchanfrage wird durch die boolsche Auswertung der Bedingungen festgestellt, ob ein Treffer vorhanden ist oder nicht.

Mustersuche
Unter Mustersuche versteht man Suchoperationen, die Abweichungen zwischen den Stichworten und den Worten im Text erlauben. Hier sind auch Wortbestandteile wie Präfix, Suffix oder Teilwörter interessant. Ein Muster kann z. B. durch einen regulären Ausdruck dargestellt werden.

---

[5] siehe [KM03]

Phrasensuche
Es kann festgelegt werden, dass die Suchbegriffe im gleichen Satz auftauchen.

Ranking
Beim Ranking werden die gefundenen Texte nach Relevanz sortiert. Bekannte Vertreter sind das Vektorraummodell und das probabilistische Modell. Bei Ersterem wird durch die Terme im Text ein Vektor berechnet. Dann wird der Abstand zwischen diesem Vektor und dem Anfragevektor berechnet. Anhand des Abstandes werden die Ergebnisse nun sortiert. Das zweite Modell berechnet die statistische Relevanz eines Textes bezogen auf die Suchanfrage.

In klassischen Information-Retrieval-Systemen werden zur Indexierung im Wesentlichen invertierte Dateien, Suffixbäume/-arrays und Signaturbäume. Deshalb werden diese Verfahren im Folgenden näher erläutert.

3.3.2 Invertierte Datei
Die invertierte Datei (invertierte Liste, inverted file) ist beim Information Retrieval die meist benutzte Technik zur Indizierung großer Textbestände. Zu jedem zu indizierenden Wort wird die Position im Dokument festgehalten. Daher kommt auch der Ausdruck *invertierte Datei*, da im Dokument zu jeder Position ein Wort gehört und dies dann den umgekehrten Fall beschreibt. Die invertierte Datei besteht somit aus Paaren von Worten und Listen mit den Positionen im Dokument. Dabei wird von einem statischen Dokumentbestand ausgegangen, und eine Änderung erfordert meist eine Änderung im kompletten Index. Will man in mehreren Dokumenten suchen, muss zusätzlich noch der Dokumentidentifikator zu jeder Position gespeichert werden.

Die Position eines Wortes kann auf zwei Arten angegeben werden:

1. Die Worte werden durchnummeriert. Hierbei wird von der *Wortposition* gesprochen. Es entsteht eine stärkere Trennung vom Dokument.
2. Es wird die Position des ersten Zeichens des Wortes innerhalb eines Dokuments gespeichert. Dies ist gut geeignet um direkt auf einem Dokument zu arbeiten. Ein Editor kann z. B. Wörter farblich hervorheben.

Abbildung 7 zeigt den Zusammenhang zwischen invertierter Datei und Dokument für die Wortposition:

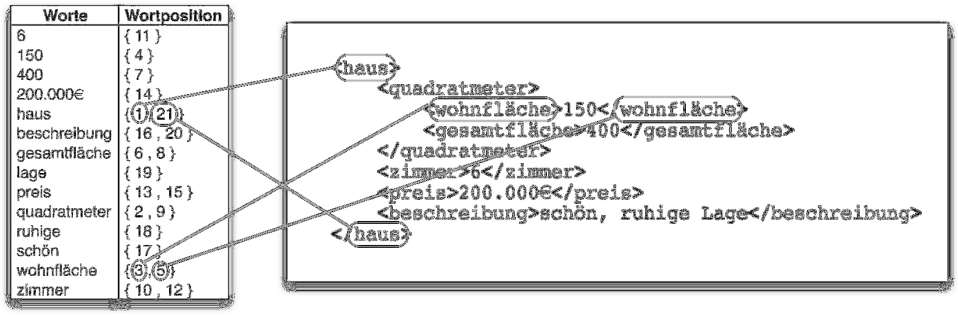

Abb 7. Zusammenhang zwischen invertierter Datei und Dokument

Speicherung
Invertierte Listen können gut mit einem B*-Baum gespeichert werden. Dabei sind die Wörter die Suchschlüssel des Baumes, und in den Blättern werden die Verweislisten abgelegt.

Stichwortsuche und boolsches Retrieval
Bei einer konjunktiven Verknüpfung wird bei den über den B*-Baum gefundenen Stichworten der Durchschnitt gebildet, bei disjunktiver Verknüpfung werden sie vereinigt und der Ergebnismenge hinzugefügt.

Mustersuche
Da der B*-Baum nur Bereichsanfragen und Präfixanfragen erlaubt, ist die Suche nach beliebigen Teilworten oder eine approximative Suche nicht möglich. Ersteres kann man erreichen, indem man auch Suffixe, n-Gramme[6] oder Rotationen eines Wortes mit indiziert, jedoch auf Kosten von Speicherplatz.

3.3.3 Tries
Tries sind für die Präfixsuche optimiert. Als Datenstruktur wird ein Baum benutzt. Alle Zeichenketten, die ein gemeinsames Präfix besitzen, befinden sich an einem gemeinsamen Knoten und haben einen gemeinsamen Teilpfad. Daraus ergibt sich, dass ein Knoten maximal so viele Kanten besitzt wie das Alphabet Elemente hat. Die Tiefe wird vom längsten Wort im Baum bestimmt, und alle gespeicherten Zeichenketten lassen sich wieder rekonstruieren. Der Vorteil liegt in einer im Allgemeinen sehr kompakten Speicherung. Falls die indizierten Worte jedoch nur einen kleinen Anteil der kombinatorisch Möglichen ausmachen, gibt es kaum Speichervorteile, da dann viele Knoten nur einen Nachfolger haben. Da Tries Hauptspeicherstrukturen sind, sollte der Index nicht zu einem großen Teil ausgelagert werden müssen.

---

6 siehe [KM03]

3.3.4 Patriciabäume

Patriciabäume sind eine besondere Form von Tries und beheben das Manko in Bezug auf den Speicherplatz. Anstelle der einzelnen Zeichen wird, um eine Verzweigung vorzunehmen, die Position des nächsten relevanten Zeichens im Knoten abgelegt. An den Kanten steht dann jeweils dieses nächste Zeichen. Daraus folgt, dass jeder Knoten nun entweder ein Blatt oder eine Verzweigung darstellt und die Pfade nicht unnötig lang werden. In den Blättern steht nur noch das verbleibende Suffix. Nachteilig ist die Tatsache, dass Patricia-Bäume nicht balanciert sind.

3.3.5 Suffixbäume

Diese Baumstruktur besteht aus allen Suffixen eines Wortes und indexiert alle Teilzeichenketten eines Wortes. Zur Konstruktion sei hier auf Ukkonens Algorithmus verwiesen [UKK95]. Um einen Teilstring zu suchen, wird der jeweils passenden Verzweigung nachgegangen, bis der Teilstring abgearbeitet ist, oder es keine passende Verzweigung gibt. Das hier vorgestellte Verfahren lässt sich leicht erweitern, um die Positionen zu erfahren, an denen die Teilstrings vorkommen. Speichereffiziente Variante der Suffixbäume sind die nun folgenden Suffixarrays.

3.3.6 Suffixarrays

Wie beim Suffixbaum unterstützen Suffixarrays die Suche nach Teilzeichenketten. Ein Vektor verweist auf jedes Suffix im Wort wie Abbildung 8 verdeutlicht. Dabei sind die Verweiseinträge lexikographisch sortiert. Eintrag 4 kommt somit vor 6, da „chtturm“ lexikographisch vor „rm“ kommt.

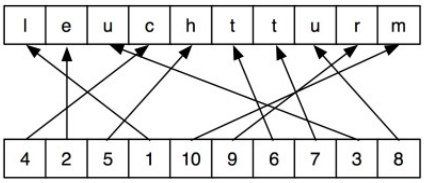

Abb. 8. Beisiel für ein Suffixarray

3.4 Pfadindex

Beim Pfadindex werden die Pfade des gerichteten, markierten Graphen eines XML-Dokuments indexiert, der in 3.1 erläutert wurde.

Zusätzlich zu den allgemeinen Kriterien, können Pfadindexe folgende Eigenschaften haben:

- Auf den Pfaden können Musteranfragen ausgeführt werden.
- Es kann nur vorwärts oder auch rückwärts im Graphen navigiert werden.
- Es kann ein Schema berücksichtigt werden.
- Häufige Suchanfragen können optimiert werden. Man spricht dann von adaptiven Indexen.

In den folgenden Abschnitten werden unterschiedliche Varianten von Pfadindexen eingeführt.

3.4.1 Index Fabric

Index Fabric ist ein sehr gut skalierender Index, dessen grundlegende Struktur Patricia-Bäume sind. Die Grundidee besteht darin, Pfade als Strings zu indizieren. Dies ist auch bei sehr großen Dokumenten gut möglich, da Index Fabric unempfindlich gegenüber der Länge der einzutragenden Schlüssel ist. Eine Anfrage wird in eine Suchzeichenkette kodiert, welche dann durch Vergleiche mit den als Strings kodierten Pfaden des Patricia-Baums abgearbeitet wird. Bei der Indexierung werden zwei Arten unterschieden: Einmal können alle Pfade indexiert werden, welche *Raw Paths* genannt werden. Zweitens können bestimmte Pfadzugriffe optimiert werden, hier spricht man von *Refined Paths.*

Da Patricia-Bäume nicht balanciert sind, kann sich das negativ auf die Performance auswirken. Um dies auszugleichen werden beim Einfügen Zugriffsebenen erstellt, die als Einstiegsstruktur für die Abarbeitung des Baums fungieren.

Abbildung 9 zeigt einen Patricia-Baum mit den zugehörigen Zugriffsebenen.

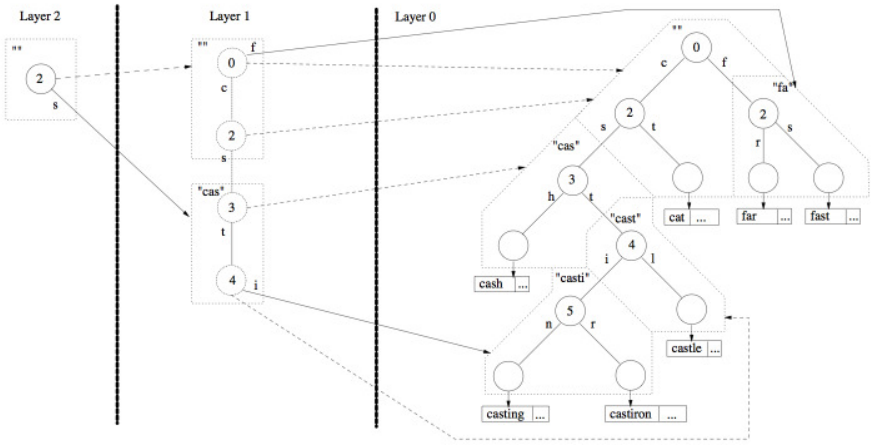

Abb. 9. Beispielaufbau des Patriciabaums mit den Zugriffsebenen

Aufbau der Ebenen und des Patricia-Baums:

- Auf Ebene 0 befindet sich der ursprüngliche Baum mit der Ausnahme, dass alle Blätter nicht direkt im Baum abgelegt werden, sondern sich auf einer seperaten Speicherungsebene befinden.

- Die Baumstrukuren sind in Teilbäume zerlegt, welche genau auf eine Seite passen.
- Die Baumstruktur auf der Ebene 1 stellt einen Index über den Präfixen der Blöcke auf der Ebene 0 dar. Diese Präfixe sind in der Abbildung in Hochkommata gesetzt.
- Alle Teilbäume einer Ebene werden anhand ihres Präfixes beschrieben und auf der darüberliegenden Ebene referenziert.
- Es gibt zwei Arten von Verweisen, die Ebenen verbinden.
- Markierte entfernte Verweise sind mit einer durchgehenden Linie dargestellt. Sie haben dieselbe Bedeutung wie in einem normalen Patricia-Baum und verbindet Knoten.
- Nichtmarkierte Direktverweise sind mit einer gestrichelten Linie dargestellt. Sie verbinden einen Knoten einer Ebene mit einem Block, der diesen Knoten auf der nächsten Ebene enthält.

Suche

Angefangen wird im Wurzelknoten des Blocks der obersten Ebene, siehe Abbildung 9 ganz links. In den Blöcken verläuft die Suche analog zum Patricia-Baum, in dem die Zeichen verglichen werden und den entsprechenden Kanten gefolgt wird. Bei einem Verweis über Blöcke hinweg wird diesem gefolgt und der entsprechende Block auf der nächsten Ebene untersucht. Falls keine markierte Kante gefunden wird, wird einer unmarkierten auf einen Block der nächsten Ebene gefolgt. Ist man auf der untersten Speicherebene, den Blättern, angelangt, muss der Wert noch erfolgreich verglichen werden, da die Knoten nur eine verlustbehaftete Darstellung des gespeicherten Wertes darstellen. Auf jeder Ebene wird stets ein Block betrachtet. Bedingt durch die verlustbehaftete Kompression kann es passieren, dass man in den falschen Block gelangt, dann muss man wieder eine Stufe nach oben. Dieser Fall tritt in der Praxis jedoch sehr selten ein. Durch die komprimierte Speicherung können sehr viele Schlüssel in einem Block gespeichert werden, was zu einem breiten, niedrigen Baum führt.

Änderungsoperationen

Einfüge-, Lösche- und Änderungsoperationen können ähnlich effizient wie die Suche ausgeführt werden. Das Einfügen erfolgt immer auf Ebene 0 und ist mit Hinzufügen eines einzelnen neuen Knotens oder Kante zu einem existierenden Knoten verbunden. Bei einem Überlauf wird die aktuelle Seite aufgeteilt und auf der Ebene darüber ein Knoten eingefügt. Dies kann sich bis zum Erstellen einer neuen Ebene fortsetzen. Beim Löschen eines Schlüssels folgt der umgekehrte Prozess.

Kodierung der Pfade

Die Elementnamen werden auf ein besonderes Alphabet (Designators) abgebildet, um eine effizientere und eindeutige Zuordnung zu erhalten. Gleiche Elementnamen werden auf unterschiedliche Bezeichner abbgebildet. Diese Sonderzeichen (oder -ketten) kann man als Schlüssel auffassen und werden in einem speziellen Verzeichnis abgelegt werden. In Abbildung 10 wird z. B. der Pfad gebäude/hochhaus/stockwerke auf “G H S” abgebildet.

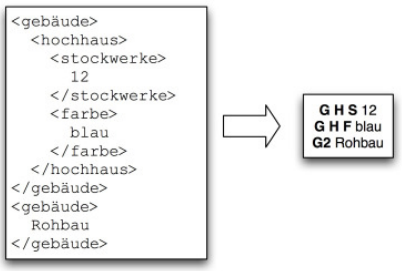

Abb. 10. Abbildung der Pfade auf Symbole

Die Pfade in den Suchanfragen werden auf die gleiche Weise kodiert, bevor nach ihnen gesucht wird.

3.4.2 SphinX

Im Gegensatz zu Index Fabric verabeitet SphinX auch die Schemainformation. Prinzipiell kann jedes Schema verabeitet werden, der Schwerpunkt wird aber auf DTD (Document Type Definition) gelegt. Zu Beginn werden zwei Graphen, der Document Graph und der Schema Graph, aus dem XML-Dokument und dem DTD-Schema erstellt. Der Document Graph ist der Graph des XML-Dokuments, bei dem die Knoten bi-direktional verknüpft sind. Auf die gleiche Weise erhält man durch Konvertieren des DTD einen Schema Graph. Für alle indexierten Pfade im Dokument
enthält jedes Blatt des Schema Graph einen Verweis auf einen B*-Baum, dessen Blätter wiederum auf die atomare Werte des Dokuments verweisen. Die B*-Bäume indexieren also die atomaren Werte des XML-Dokuments. Abbildung 11 veranschaulicht diesen Zusammenhang noch einmal.

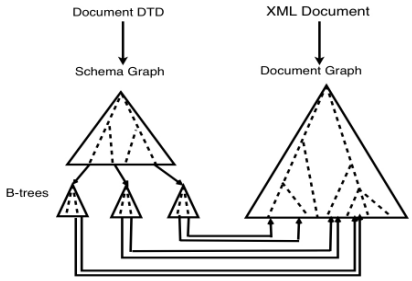

Abb. 11. Übersicht Document-Graph – Schema-Graph

Document-Graph

Der Dokument-Graph bildet das XML-Dokument auf einen gewurzelten, markierten Graphen ab. Die Knoten entsprechen den Elementen und die Blätter den atomaren Werten. Durch die bi-direktionale Verknüpfung der Knoten können die Eltern und Kinder eines Knotens schnell gefunden werden. Abbildung 12 zeigt einen Document-Graph für ein Beispieldokument.

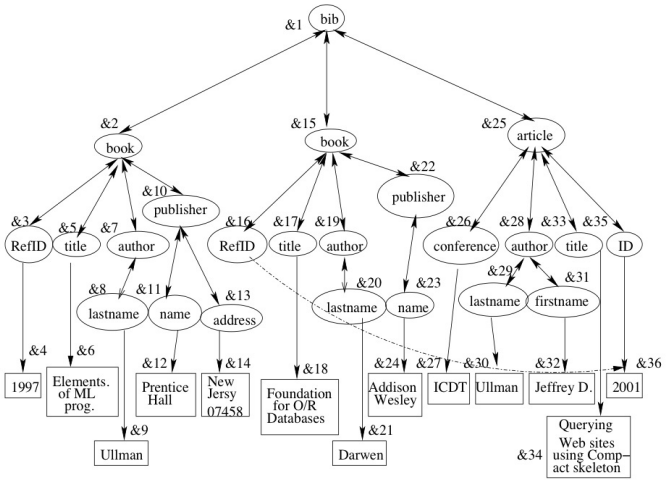

Abb. 12. Beispiel-Document-Graph

Schema-Graph
Hier korrespondieren die Knoten mit den Elementen und Attributen des DTD. Die Kanten bilden die inhaltliche Zugehörigkeit der Elemente ab und sind gerichtet. Durch rekursive Verweise im DTD ensteht im Allgemeinen ein Graph.
Um die Erzeugung des DTD-Graphen zu beschleunigen, werden Quantifikatoren (*, +, ?) ignoriert, und Alternativen (a | b) werden zu (a , b) umgewandelt. Durch diese Vereinfachung erreicht man die Konstruktion des Graphen mit linearem Aufwand. Ähnlich wie beim Patricia-Baum ist dies mit Informationsverlust verbunden, da der so modifizierte DTD mehr gültige Dokumente beschreiben kann als vorher.

B-Bäume des Schema-Graph*
Es verweisen nicht alle Pfade des Schema Graph auf einen B*-Baum. Die Bäume werden anhand des Anfrageaufkommens bestimmter Pfade oder eines anderen Kriteriums erstellt, damit die Speicherkosten nicht zu groß werden.

Suche
Die Suchanfragen werden in drei Schritten bearbeitet.

Der erste Schritt wandelt eventuelle allgemeine Pfadausdrücke in eine Menge von einfachen um und testet, für welchen ein Index im Schema Graph existiert. Im zweiten Schritt werden die mit den Selektionsprädikaten aus der Suchanfrage korrespondierenden Schlüssel durch die vorher erhaltenen B*-Bäume ausgewertet. Dadurch entsteht eine Menge von Verweisen auf die Knoten des Document Graph. Bei der

Traversierung des Document Graph werden nun durch einfache Navigation, entsprechend der Suchanfrage, die passenden Teildokumente gefunden.

3.4.3 APEX-Index

APEX ist ein adaptiver Index, der die Suchanfragen analysiert, und nur häufige Anfragen indexiert. Grundlage um die XML-Struktur zu berücksichtigen ist wieder ein gerichteter, markierter Graph, der G_{APEX} genannt wird. In jedem Knoten steht die Ordnung des Elements im XML-Dokument und die Kanten korrespondieren zu Beziehungen zu Elementen oder Verweise auf andere Elemente. Um eine Unterscheidung zwischen Beziehung und Referenz zu erhalten, starten Referenzen mit einem '@'. Als zweite Struktur ist eine Kombination aus Baum und Hashtabelle vorhanden, H_{APEX}, mit der die Knoten in G_{APEX} angesprungen werden können.

Struktur H_{APEX}:

- Die Knoten von $H_{APEX,}$ *hnodes*, enthalten eine Hashtabelle, deren Einträge entweder wieder auf einen *hnode* oder auf einen Knoten in G_{APEX} verweisen.
- Jeder Knoten in G_{APEX} wird auf einen Eintrag in einem *hnode* abgebildet.

Folgende Abbildung zeigt einen Beispielaufbau von H_{APEX} und G_{APEX}:

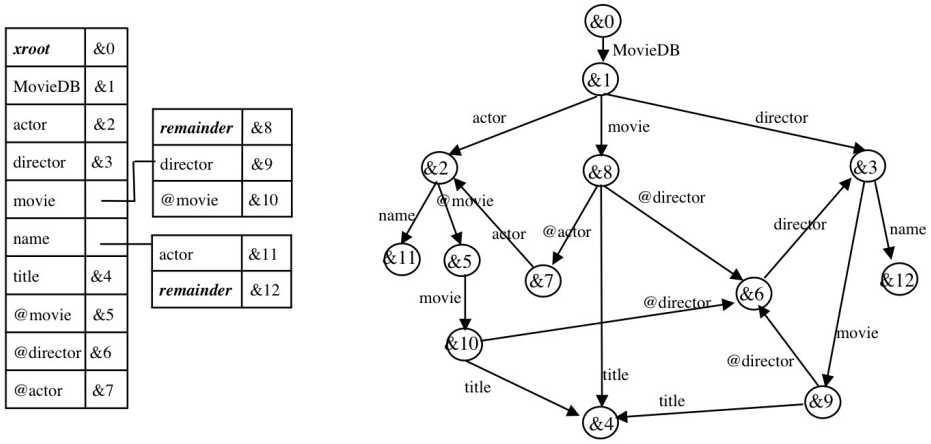

Abb. 13. Beispiel für APEX. Links sieht man H_{APEX} und rechts G_{APEX}.

3.4.4 A(k)-Index

Der A(k)-Index reduziert die Speicherkosten, indem er nicht alle Pfade in einem Graphen indiziert. Es wird die Ähnlichkeit Knoten und der Pfade zu ihnen ausgenutzt. Dadurch können Knoten zusammengefasst und somit Speicherkosten reduziert werden. Durch die Zusammenfassung wird der Graph allgemeiner, und damit können mehr Pfade möglich sein als in der ursprünglichen Darstellung. Der A(k)-Index wurde nicht speziell für XML entwickelt, und ist allgemein geeignet für Daten, die eine Graphenstruktur aufweisen.

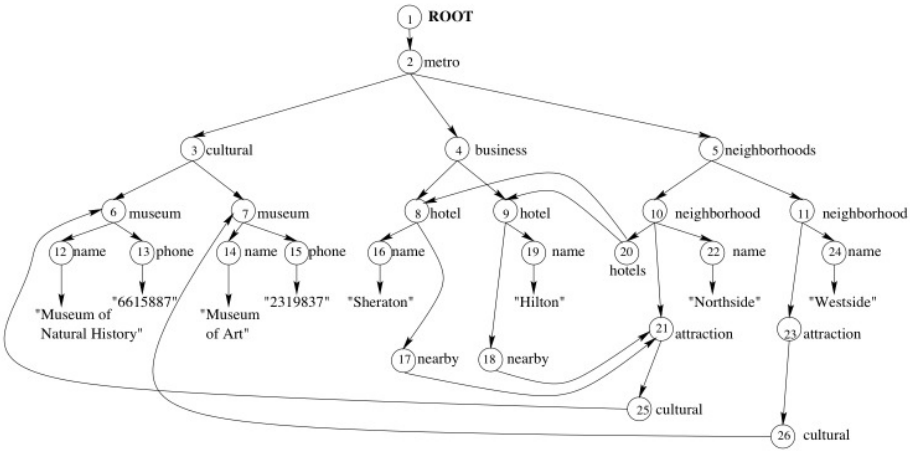

Abb. 14. Beispielgraph

Grundlagen

- bisimilar: zwei Knoten sind bisimilar, wenn alle Pfade zu ihnen gleich sind. In obiger Abbildung sind demnach 8 und 9 bisimilar, was bei 21 und 23 nicht zutrifft.
- k-bisimilar ($\approx^k$)
  - Für zwei Knoten u,v gilt genau dann u $\approx^k$, wenn es für jeden Elternknoten u' von u einen Elternknoten v' von v gibt, so dass u' $\approx^k$ v' gilt und umgekehrt.
  - Es gilt genau dann u $\approx^0$ v, wenn sie die gleiche Markierung haben.

Aufbau

Grundidee ist, ähnliche Knoten (wie z. B. die museum-Knoten in Abb.14) zusammenzufassen. Als Ähnlichkeitsbeziehung wird $\approx^k$ benutzt. Mit dieser Beziehung werden ähnliche Knoten zusammengefasst und man erreicht, dass bis zur Pfadlänge k nicht mehr Pfade möglich sind als im ursprünglichen Graphen. Die ursprünglichen Kanten werden mit den neuen Knoten verknüpft. Abbildung 15 zeigt die Veränderung des Anfangsgraphen bei zunehmendem k. Ab einem bestimmten k beschreibt der A(k)-Index alle möglichen Pfade.

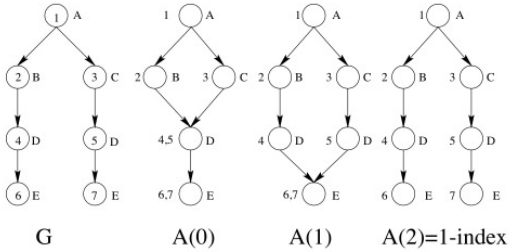

Abb. 15. Änderung im Aufbau des A(k)-Index bei Erhöhen von k

3.5 Baumbasierte Indexierung

Bei der Baumbasierten Indexierung werden die Knoten speziell nummeriert, um die Baumstruktur abzubilden. Um den Vaterknoten eines Knotens zu berechnen, nutzt man folgende Formel:

$$Parent(i) = \frac{i-2}{k} + 1$$

Sie gilt für einen Baum bei dem alle Knoten, außer die Blätter, k Kinder besitzen. Die Knotennummern werden *UIDs* (Unique element IDentifier) genannt. Eine Implementierung wird in dem hybriden Index BUS in Kapitel 3.6.3 beschrieben.

3.6 Hybride Verfahren

In realen Systemen zur XML-Indexierung müssen Anfragen verarbeitet werden, die sich auf den Inhalt und die Struktur beziehen. Im Folgenden werden einige Möglichkeiten, dies zu erreichen, beschrieben.

3.6.1 XML-sensitive Volltextsuche

Durch das Speichern der Elementnamen in einer separaten invertierten Liste kann mit der Volltextsuche auf die Subelementstruktur eines XML-Dokuments eingegangen werden. Anfragen der Form hotel // name sind so möglich. Dieses Verfahren kann man erweitern, in dem man die Ordnung und Verweise auf Vorgänger mit in die invertierte Datei aufnimmt. Man hat dann einen normalen Volltextindex des Dokuments, und einen Zweiten, welcher die Struktur speichert.

3.6.2 Pfadindexierung und Volltextindex

Hier werden ein Pfadindex und ein XML-sensitiver Volltextindex verwendet. Beim Volltextindex stehen in der invertierten Liste die KnotenIDs des Elements, in der das Wort vorkommt, anstatt der Position. Bei einer Anfrage liefern beide Verfahren eine Menge von Knoten, die gemäß der Anfrage zusammengeführt werden.

3.6.3 BUS (Bottom-Up-Scheme)

Bus kombiniert Struktur und Inhalt, und zusätzlich erfolgt ein Ranking der Ergebnisse durch die Häufigkeit, mit der ein Suchwort vorkommt.

Idee

BUS nutzt ein auf der baumbasierten Indexierung aufbauendes Verfahren, um die Struktur des XML-Dokuments zu indexieren. Grundlage ist das UID-Verfahren zur Nummerierung von Knoten eines Baumes. Bei diesem Verfahren muss jeder Knoten die gleiche Anzahl an Kindern haben. Um dies zu erreichen, wird die Maximalzahl an direkten Nachfolgern eines Knotens im XML-Dokument bestimmt, und alle anderen Knoten erhalten zusätzlich virtuelle Kinder, um die gleiche Anzahl zu bekommen. Es werden nicht wirklich neue Kinder hinzugefügt, aber bei der Nummerierung werden diese Kinder dann mitgezählt und man kann mit folgender Formel schnell den Eltern-

knoten eines Elements berechnen (ParentID = UID des Elternknotens, i = aktuelle UID, k = maximale Knotenanzahl):

$$Parent(i) = \frac{i-2}{k} + 1$$

Abbildung 16 verdeutlicht die Nummerierung:

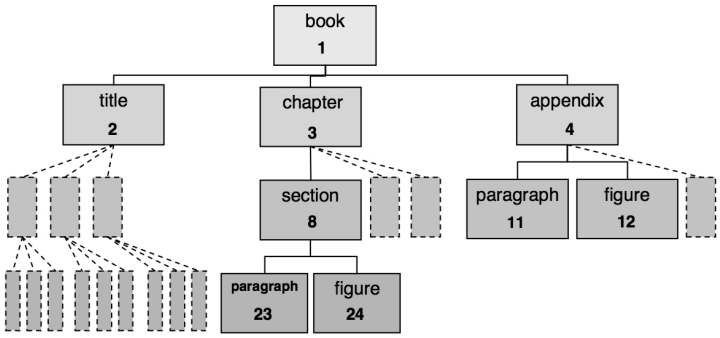

Abb. 16. Nummerieung eines XML-Dokuments mit UIDs

Durch eine solche Kodierung erhält man jedoch keinen Aufschluss über den Elementtyp in den Nummern. Deswegen wird bei BUS die GID-Nummerierung (General Element Identifier) eingeführt. Bei GID besteht eine Knotennummer aus 4 Bestandteilen:

- Dokumentnummer,
- UID des Elements,
- Level, auf dem sich das Element befindet,
- Elementtyp, kodiert als Zahl.

Abbildung 17 zeigt den Baum mit der neuen Kodierung.

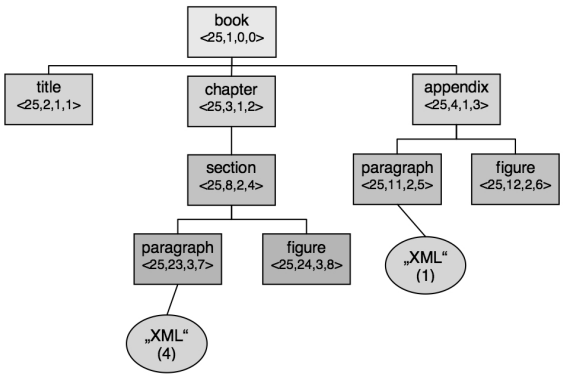

Abb. 17. Kodierung des Dokuments mit GIDs

Konstruktion des Index
Als Erstes wird jedem Knoten ein GID nach obigem Verfahren zugewiesen. Danach wird in einer invertierten Liste zu jedem Wort des Dokuments (nur der Textinhalt wird betrachtet) der zugehörige GID und die Häufigkeit des Wortes in diesem Element gespeichert. Implementiert wird die invertierte Datei als B*-Baum, wie in Kapitel 3.3.3 erwähnt.

Suche
Bei einer Suche wird die Anzahl der Treffer in Akkumulatoren[7] für das nachfolgende Ranking zwischengespeichert. Zuerst wird im Schema überprüft welcher Elementtyp für die Anfrage relevant ist. Für die Anfrage
`/book/chapter/secton[„XML“]`
ist das Typ 4, 7 und 8 wenn man den Baum aus Abbildung 17 als Grundlage nimmt. Nun werden alle Einträge in der invertierten Datei mit Elementtyp 7 oder 8 für das Wort „XML“ gesucht. In unserem Fall ist das „XML <4<25,23,3,7>>“. Über die Formel zur Berechnung des Elternknotens wird der UID des Vaterknotens berechnet. In diesem Fall ist das (23-2)/3+1 = 8. Für dieses Element wird in der Akkumulatortabelle die Häufigkeit des Wortes „XML“ in Element 23 hinzuaddiert. Da sie vorher 0 war erhält man 4. Um die Ergebnisse der Relevanz nach zu ordnen wird noch die Dokumentfrequenz berechnet. Hierfür wird auf [SJJ] verwiesen.

3.7 Zusammenfassung

Bei der Indexierung eines XML-Dokuments kann man eine Trennung in Werte-, Volltext-, Strukturindexe und hybride Verfahren vornehmen. Nachfolgend werden die vier Arten zusammenfassend dargestellt:

- Ein *Werteindex* erlaubt den Zugriff auf atomare Werte eines XML-Dokuments. Darunter fallen z. B. Elementnummern, Attribute oder Werte eines bestimmten Datentyps (z. B. Integer, wird im Schema festgelegt).
- Ein *Volltextindex* indiziert die einzelnen Worte des XML-Dokuments. Die Auswertung erfolgt mit Information-Retrieval-Techniken.
- *Strukturindexe* sehen ein XML-Dokument als Baum oder Graph und können so den speziellen Aufbau eines Dokuments berücksichtigen. Eine effiziente Auswertung von Anfragen, die die Dokumentstruktur mit einbeziehen, ist damit möglich.
- *Hybride Verfahren* ermöglichen Anfragen, die sich auf die Struktur und den Inhalt eines Dokuments beziehen. Dies wird durch Kombinationen aus Vertretern der vorigen Arten erreicht.

[7] Einträge in Form einer Tabelle

4. Fazit

Durch die Popularität des XML-Formats und seines breiten Anwendungsspektrums müssen entsprechende Programme, Dienste und insbesondere Datenbanken mit großen Datenmengen arbeiten können. Dabei benötigt man Indexierungstechniken, um effizient auf den Daten zu arbeiten. Da XML-Dokumente Struktur und Inhalt kombinieren, dabei aber keinem festen Schema folgen, müssen alte und neue Ansätze bei der Indexierung zusammengeführt werden. XML ist ein noch recht junger Standard, und somit ein aktuelles Forschungsgebiet, speziell im Hinblick auf Indexierungsverfahren. Es existieren aber eine Fülle von Verfahren, von denen einige sehr gut getestet und implementiert sind. Außerdem gibt es bereits kommerzielle Anbieter, die XML-Datenbanken anbieten, und damit XML-Indexierung in diesen Systemen umgesetzt haben. Zu nennen sind hier *DB2 9 with pureXML technology* von IBM, *Infobyte-DB* als Ausgliederung des Fraunhoferinstituts IPSI oder die Berkley DB XML von Oracle. Dennoch sind XML-Datenbanken oft noch nicht derart ausgreift wie z.B. relationale Datenbanken, vor allem in Bezug auf Speichermanagement, Breite der möglichen Suchanfragen oder Effizienz. Außerdem wäre es wünschenswert, die vorhandenen Indexierungsverfahren unabhängig zu bewerten - dies wird ist bis jetzt nur unzureichend der Fall. Meist wird nur von der Seite, die ein Verfahren erweitert oder verbessert, ein Vergleich geboten.

Ein positiver Nebeneffekt, bedingt durch die starke Forschung, ist eine große Auswahl an aktuellen Projekt-, Diplom- und Doktorarbeiten, die sich dem Thema widmen.

Abschließend kann man sagen, dass derzeit intensiv nach guten Verfahren zur XML-Indexierung geforscht wird – jedoch wird noch einige Zeit verstreichen, bis die Verfahren so ausgereift sind wie die klassischen Varianten bei z. B. relationalen Datenbanken.

5. Literaturnachweis

[BLK04] Lidiya Buda, Iryna Laxhuber, Oxana Koriakina. BUS - Ein effizientes Indexierungs- und Retrieval-Schema für strukturierte Dokumente. Centrum für Informations- und Sprachverarbeitung Ludwig-Maximilians-Universität München (2004)

[CMK02] Chin-Wan Chung, Jun-Ki Min, Kyuseok Shim. APEX: An Adaptive Path Index for XML Data. Div. of Computer Science, Taejon. College of Engineering, Seoul National University. Korea. (2002)

[CSF01] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason1, M. Shadmon. A Fast Index for Semistructured Data. Department of Computer Science, Department of Computer Science. (2001)

[HÄR] T. H. Härder. Grundlagen betrieblicher Informationssysteme. Skript.

[KM03] Meike Klettke, Holger Meyer. XML & Datenbanken – Konzepte, Sprachen und Systeme. dpunkt.verlag (2003). 155-248.

[KSB] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. University of Wisconsin. University of Washington. Bell Laboratories. Ben-Gurion University.

[LAU05] Georg Lausen. Datenbanken – Grundlagen und XML-Technologien. Spekrum Akademischer Verlag (2005). 206-214.

[LH04] K. P. Leela, J. R. Haritsa. SphinX: Schema-Conscious XML Indexing. Indian Institute of Science, Bangalore, India. (2004)

[LLD02] Robert W.P. Luk, H.V. Leong, Tharam S. Dillon, Alvin T.S. Chan, W. Bruce Croft, James Allan. A Survey in Indexing and Searching XML Documents. Center for Intelligent Information Retrieval, Computer Science Department, University of Massachusetts. Department of Computing, Hong Kong Polytechnic University, Hong Kong. (2002)

[MHS] C. Mathis, T. H. Härder, K. Schmidt. A Unified Approach to Content, Structure, and Combined Indexing of XML Documents. Dept. of Computer Science, University of Kaiserslautern, Germany.

[SJJ] D. Shin, H. Jang, H. Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. Department of Computer Engineering, Chungnam National University, Taejon.

[SY06] T. Shimizu, M. Yoshikawa. Full-Text and Structural Indexing of XML Documents on B+-Tree. (2006)

[UKK95] E. Ukkonen. On-line construction of suffix trees. Algorithmica. (1995). 249-260