XML-Ähnlichkeitssuche

Christian Bering Carsten Greiveldinger Regis Newo

July 2003

Inhaltsverzeichnis

1 Vorwort **1**

2 ATreeGrep: **1**
2.1 Einleitung . . . 1
2.2 Problemdefinition . . . 1
2.3 Das Approximate nearest neighbor search problem . . . 1
2.4 Lösung ohne Wildcards in der Query . . . 2
2.4.1 Aufbau der Suffix-Array-Datenbank . . . 3
2.4.2 Die Suche in der Sufix-Array-Datenbank . . . 3
2.5 Erweiterung der Query mit Wildcards . . . 4
2.6 Verbesserte Lösung mit Filtern . . . 5
2.7 Experimente und Ergebnisse . . . 5
2.8 Fazit . . . 7

3 Ähnlichkeitssuche mittels Relaxationen in Join-Plänen **7**
3.1 Einleitung . . . 7
3.2 Problemstellung . . . 8
3.3 Suchmuster als Join-Plan . . . 9
3.4 Relaxationen . . . 9
3.5 Gewichtung und Bewertung . . . 12
3.6 Evaluierung des Join-Plans . . . 13
3.6.1 Thres . . . 14
3.6.2 Opti-Thres . . . 14
3.6.3 Top-K . . . 15
3.7 Experimentelle Ergebnisse . . . 15
3.8 Zusammenfassung und Diskussion . . . 16

4 Unscharfe Joins über XML-Daten **16**
4.1 Einführung und Anregung . . . 16
4.2 Editierdistanz . . . 18
4.2.1 Editierdistanz für Strings . . . 18
4.2.2 Editierdistanz für Bäume . . . 18
4.3 Problemstellung . . . 19
4.4 Exakte Schranken für die Editierdistanz . . . 19
4.4.1 Untere Schranke . . . 19
4.4.2 Obere Schranke . . . 20
4.5 Geschätzte Schranken . . . 20
4.6 Algorithmen . . . 21
4.6.1 Der naive Algorithmus (N) . . . 21
4.6.2 Algorithmus mit exakten Schranken (B) . . . 21
4.6.3 Algorithmus mit geschätzten Schranken (RS) . . . 22

4.6.4 Algorithmus mit beiden Schranken (RSB) 22
4.6.5 Algorithmus mit geschätzten Distanzen zur Referenzmenge (RSC) 22
4.7 Experimentelle Evaluierungen 22
4.7.1 Evaluierung der exakten Schranken 22
4.7.2 Evaluierung der Algorithmen 23
4.7.3 Evaluierung über DBLP 23
4.8 Zusammenfassung 23

1 Vorwort

Diese Ausarbeitung ist im Rahmen des Seminars XML-Technologien von Dr.-Ing. Ralf Schenkel im Sommersemester 2003 an der Universität des Saarlandes entstanden. Der Seminarvortrag wurde am 22. Juli 2003 von Christian Bering, Carsten Greiveldinger und Regis Newo gehalten und stellt drei Artikel mit verschiedenen Ansätzen zur Problematik der Ähnlichkeitssuche auf XML-Daten dar. Im ersten wird das Suchproblem mittels Stringmatching-Ansätzen gelöst (ATreeGrep aus [SWSZ02]), der zweite Ansatz ([AYCS02]) formuliert es im Rahmen einer Join-Architektur, und der dritte Artikel ([GJK+02]) stellt effiziente, generelle Variationen für Join-Algorithmen vor.

2 ATreeGrep:

XML-Ähnlichkeitssuche als Stringmatching-Problem

2.1 Einleitung

AtreeGrep ist eine Anwendung, die mit dem Ziel erarbeitet wurde eine Lösung für das Problem der Ähnlichkeitssuche auf ungeordneten Bäumen zu finden. Dies ist ein weit verbreitetes Problem in verschiedenen Bereichen der Wissenschaft, nicht nur in der Informatik, sondern unter anderem auch in der Biologie. Die Motivation für „ATreeGrep“ war, daß dieses Problem sowohl in der Handhabung mit XML-Daten in der Informatik, als auch in der Molekularbiologie bei der Speicherung von Molekülketten auftrat.
Im Folgenden wird zuerst das Problem spezifiziert, anschließend eine einfache Lösung präsentiert, die noch keine Wildcards in der Query unterstützt, dann eine erweiterte Lösung mit Wildcard-Option, daraufhin eine Verbesserung des einfachen Ansatzes mit Hilfe einer Heuristik, und schließlich die Präsentation von Tests und deren Auswertung.

2.2 Problemdefinition

Das Problem stellt sich in der Praxis folgendermaßen dar. Man hat eine große Datenbank D mit XML-Daten, und eine Query Q, die auf den Daten ausgewertet werden soll. Die Egebnisse sollten nun möglichst ähnliche Dokumente D_i der Datenbank sein, die sich nicht mehr von der Query Q unterscheiden, als es der Benutzer mit einer vorher eingegebenen Grenze DIFF bestimmt hat.
Dies ist in der Wissenschaft keine neue Fragestellung und führt zu einem allgemeinen Problem, dem Approximate Nearest Neighbor Search Problem, kurz ANN.

2.3 Das Approximate nearest neighbor search problem

Das Approximate Nearest Neighbor Search Problem definiert sich wie folgt:
Man hat eine Query Q, die der Benutzer stellt, eine Datenbank D mit den XML-Daten D_i, in denen gesucht werden soll, und eine obere Schranke DIFF, mit der der Benutzer definiert, wie stark sich die Ergebnisse maximal von seiner gestellten Query Q unterscheiden dürfen. Man sucht nun alle Daten D_i in der Datenbank D, die der Query Q ähnlich sind, sich aber von ihr nicht mehr als die eingegebene obere Schranke DIFF unterscheiden.

Die erste Idee ist nun, die Query mit jedem Datenbaum in der Datenbank zu vergleichen und zu überprüfen, wie weit sie sich von dem betrachteten Datenbaum unterscheidet. Dafür benötigt man ein Distanzmaß, das möglichst schnell und unkompliziert berechnet werden kann, da es pro Datenbaum in der Datenbank einmal berechnet werden muß. Als ersten Ansatz würde man die Tree Editing Distanz hierfür verwenden, was aber in diesem Fall keine gute Entscheidung ist, da das Berechnen der Tree Editing Distanz auf ungeordneten Bäumen ein NP-vollständiges Problem ist. Benötigt wird ein neues und unkomplizierteres Distanzmaß. In diesem Fall entschieden sich die Autoren des zugrundeliegenden Papers als neues Distanzmaß die Anzahl der Root-to-Leaf-Pfade der Query zu zählen, die nicht in dem betrachteten Datenbaum vorkommen. In der folgenden Abbildung 1 ist ein kleines Beispiel für das eben beschriebenene Distanzmaß.

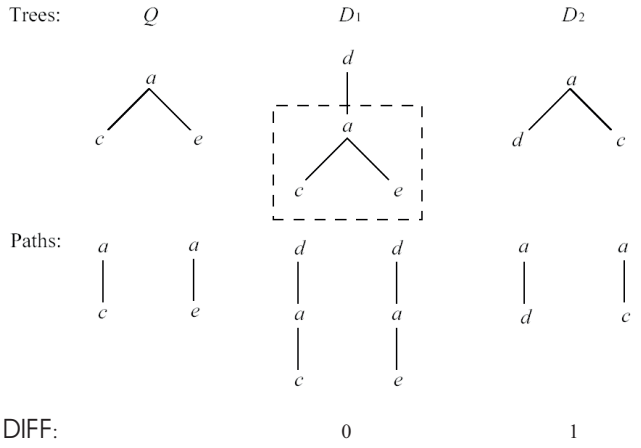

Abbildung 1: Beispiel für das Distanzmaß

Da der erste Beispieldatenbaum D_1 beide Querypfade enthält, ist die Differenz zwischen ihm und der Query Q gleich Null. Der zweite Datenbaum D_2 enthält jedoch den zweiten Querypfad nicht, woraus sich ergibt, daß er sich von der Query um eins unterscheidet, also die Differenz zwischen ihm und der Query eins ist.

2.4 Lösung ohne Wildcards in der Query

Der Basisalgorithmus, „Pathfix“ genannt, besteht aus zwei Phasen. In der ersten Phase wird aus jedem XML-Baum D_i in der Datenbank D ein Suffix-Array S_i erstellt, indem man die Root-to-Leaf-Pfade von D_i aneinanderschreibt und aus dem so entstandenen String St_i ein Suffix-Array erstellt. Dies macht man für jeden XML-Baum in der Datenbank. Das Ergebnis ist eine globale Suffix-Array-Datenbank SD, auf der im folgenden gearbeitet wird.

In der zweiten Phase, der eigentlichen Suche, werden die Root-to-Leaf-Pfade der Query Q mit denen in der Suffix-Array-Datenbank SD verglichen, um ungefähre Treffer zu finden.

2.4.1 Aufbau der Suffix-Array-Datenbank

Ein Suffix-Array ist eine Datenstruktur, die das effiziente Suchen in Strings unterstützt. Dabei ist ein Suffix ein Teilstring, der an einer bestimmten Stelle des zugrundeliegenden Strings beginnt, und mit dem Ende des Strings endet. Die Position, an der der Teilstring beginnt, wird in lexikographischer Ordnung in dem Suffix-Array gespeichert. Folglich ist ein Suffix-Array ein Array mit Pointern, die auf den Anfang eines jeden Suffixes des ursprünglichen Strings in lexikographischer Ordnung zeigt.
In diesem speziellen Fall wird das Suffix-Array aus dem String erstellt, der entsteht, wenn man die Root-to-Leaf-Pfade der Datenbäume D_i durch ein Trennzeichen „#“ getrennt hintereinander schreibt. Von dem so entstandenen String wird nun ein Suffix-Array erstellt und gespeichert.
Abblidung 2 gibt ein einfaches Beispiel für die Erstellung eines solchen Suffix-Arrays.

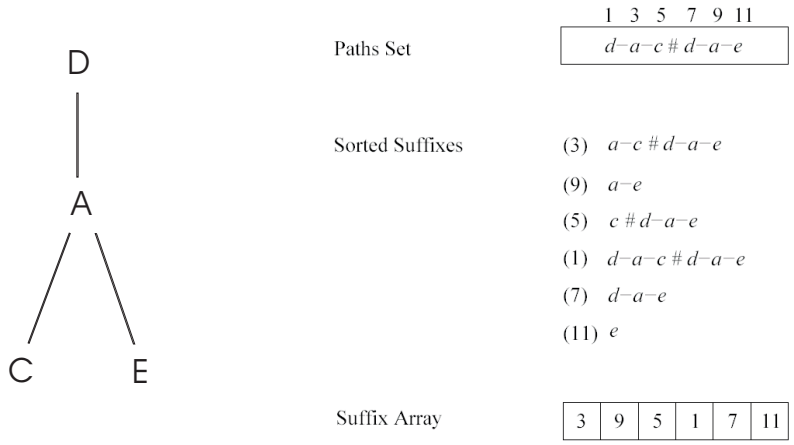

Abbildung 2: Beispiel für die Entstehung eines Suffix-Arrays

Der dargestellte Datenbaum hat die Pfade „d-a-c“ und „d-a-e“, die mit dem Trennzeichen „#“ zu einem String zusammen gesetzt werden. Darunter erkennt man alle Suffixe des Strings in lexikographischer Ordnung untereinander. Die in Klammern gestellte Zahl vor den Suffixen ist der Index, an dem der Suffix im String beginnt. Unten in der Graphik sieht man das so entstandene Suffix-Array, das nur noch die Positionen der Suffixe in lexikogaphischer Ordnung enthält.
Die Komplexität dieser Aufbauphase ist $O(MN^2)$, wobei N die Anzahl der Knoten in einem Datenbaum, und M die Anzahl der Datenbäume D_i in der Datenbank D ist. Dies ergibt sich, da jeder Datenbaum höchstens O(N) Pfade hat, und folglich maximal $O(N^2)$ Pointer für die Suffixe der Pfade benötigt werden. Dies ist aber nur der theoretische Worstcase, da Bäume mit N Knoten in der Praxis wenige Pfade oder geringe Tiefe haben. Die Komplexität ist in der Praxis also eher linear, also etwa O(MN).

2.4.2 Die Suche in der Sufix-Array-Datenbank

In der Suchphase wird die Query Q mit jedem Datenbaum D_i verglichen, wobei man eine maximale Differenz von DIFF zwischen D_i und Q zuläßt. Wird diese Schranke DIFF unterschritten, so zählt D_i zu den Ergebnissen der Anfrage. Das

Vergleichen der Query mit den Datenbäumen geschieht auf den Suffix-Arrays, indem zuerst die Wurzeln der Root-to-Leaf-Pfade von Q gesucht werden. Wenn man die Wurzel der Pfade gefunden hat, so überprüft man, ob der gesamte Query-Pfad darunterhängt. Wenn die Wurzel alle Querypfade unter sich hat, so ist die Differenz zwischen Query und Daten Null, fehlt ein Pfad, so ist sie eins, usw. Geschwister oder Eltern, die nicht Teil der Query sind, können ohne Kosten gelöscht werden. Wenn die Differenz zwischen Query und Daten DIFF nicht überschreitet, so wird der Datenbaum zur Ergebnismenge hinzugefügt. In Abbildung 3 ist die Suchphase von Pathfix dargestellt.

```
Berechne Root-to-Leaf-Pfade von Q
Für jeden Datenbaum D_i in D {
    Für jeden Pfad p in Q {
        Finde die Menge der passenden Pfade
        Wenn DIFF-Bedingung überschritten, verlasse die Schleife }
    Für jeden solchen Pfad {
        Wenn DIFF-Bedingung eingehalten
        Füge D_i zu Ergebnismenge hinzu } }
```

Abbildung 3: Die Suchphase von Pathfix

Die Laufzeit der Suchphase von Pathfix beträgt $O(q^2 * \log S)$, wobei q die Anzahl der Knoten in der Query und S die Größe eines SuffixArrays ist. Da q gleichzeitig eine obere Schranke für die Anzahl der Pfade der Query als auch für deren Länge ist und man für das Suchen eines Pfades der Länge q die Zeit $O(q*\log S)$ benötigt, folgt die obige Gesamtlaufzeit sofort.

2.5 Erweiterung der Query mit Wildcards

Da es in der Praxis oftmals so ist, daß der Benutzer nicht die genaue Dokumentstruktur kennt, wäre es von Vorteil, wenn man in die Queries sogenannte Wildcards einbauen könnte. Diese Wildcards sind Platzhalter, die entweder für einen einzigen Knoten, oder für einen ganzen Pfad stehen können.
Hier gibt es zwei verschiedene Möglichkeiten. Zum einen das „?“, welches für einen Knoten in den Datenbäumen steht. Zum anderen das „*“, welches für einen Pfad der Länge Null oder länger im Datenbaum steht. Um Pathfix auf Queries mit Wildcards anwenden zu können, wird die Query an den Wildcards aufgesplittet in mehrere Teilqueries. Anschließend wird für diese Teilqueries nach Treffern in den Daten gesucht, indem man „Pathfix“ auf die Teilqueries anwendet. Wenn man nun in einem Datenbaum entsprechende Treffer für die Teilqueries gefunden hat, muß man überprüfen, ob dabei nicht die DIFF-Schranke schon überschritten wurde. Falls dies der Fall war, gehört der Datenbaum nicht zur Ergebnismenge. Andernfalls setzt man die gefundenen Datenbaumteile zusammen und testet, ob die fehlenden Stücke zwischen den Teilbäumen mit den Wildcards der Query zu ersetzen sind. Ist dies der Fall, so wird die DIFF-Bedingung für den gesamten Baum getestet und dementsprechend der Baum zur Ergebnismenge hinzugefügt oder nicht.

In Abbildung 4 erkennt man die Teilqueries und ihre Treffer in dem betrachteten Datenbaum D sowie die Teile des Datenbaums, die von den Wildcards ersetzt werden, durch die Verbindungen mit den gestrichelten Linien zwischen Querybaum Q und Datenbaum D. Für das „?“-Wildcardsymbol wird „o“ und für das „*“-Wildcardsymbol wird der Pfad „h-j“ ersetzt. Der Teil des Datenbaums, der

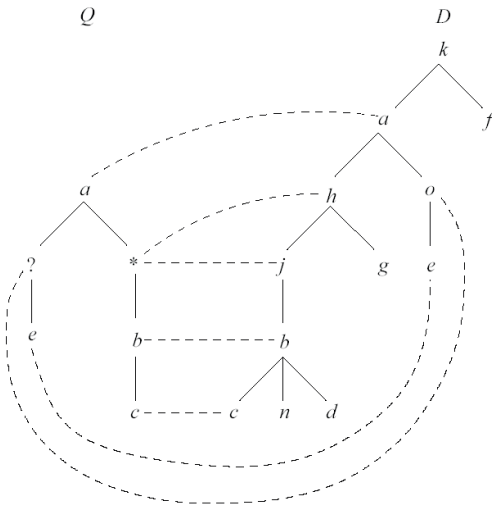

Abbildung 4: Das Vorgehen bei Queries mit Wildcards

oberhalb der Querywurzel „a“ im Datenbaum D liegt, sowie die Teile, die nicht in der Query vorkommen und außerhalb von Querypfaden liegen, werden ohne Kosten gestrichen. Damit liegt in diesem Fall ein exakter Treffer vor.

2.6 Verbesserte Lösung mit Filtern

Der bisher beschriebene Suchprozeß kann durch ein heuristisches Verfahren verbessert werden, das auf den wildcardfreien Teilen von Q arbeitet.
Es werden alle Knotenbeschriftungen und alle Vater-Kind-Paare in je einer Hashtabelle gespeichert, in der diese mit den Datenbäumen D_i, in denen sie vorkommen, verknüpft werden.
Bei der Suche nimmt man die Knotenbeschriftungen von Q und überprüft, welche D_i diese enthalten, mit maximal DIFF fehlenden Beschriftungen. Das gleiche wird mit den Vater-Kind-Paaren von Q gemacht, immer unter Beachtung der DIFF-Bedingung. Diese Heuristik eliminiert irrelevante Datenbäume D_i von der weiteren Betrachtung und zeigt eine Menge von möglichen Kanditaten auf, die im weiteren Suchverlauf genauer auf Treffer geprüft werden müssen, was dann mit „Pathfix“ geschieht.

In dem einfachen Beispiel in Abbildung 5 sieht man die beiden Hashtabellen, einmal für die Knotenbeschriftungen und einmal für die Vater-Kind-Paare, sowie die Query Q und die DIFF-Bedingung, die hier Null ist. Bei diesem einfachen Beispiel bleibt nach der Überpüfung der DIFF-Bedingung nur noch D_1 als möglicher Kandidat, da nur D_1 die Knotenbeschriftungen „a“ und „b“ und „c“ enthält, sowie beide Querypfade. Auf D_1 muß noch „Pathfix“ angewendet werden, um zu testen, ob es wirklich ein Resultat ist. Dieser Algorithmus, der sowohl die Heuristik als auch „Pathfix“ enthält, heißt „AtreeGrep“.

2.7 Experimente und Ergebnisse

Für die hier beschriebenen Experimente wurden 1000 Bäume zufällig generiert, von denen jeder 100 Knoten hatte. Die Anzahl der Pfade in einem Baum schwankte

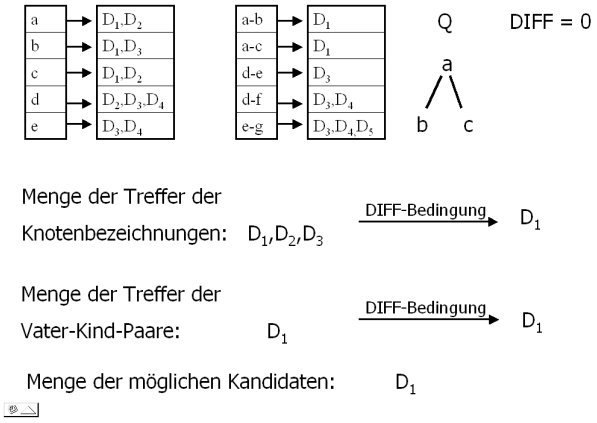

Abbildung 5: Beispiel für Filtern

zwischen 23 und 62, und die Länge der Pfade zwischen 2 und 11. Die Knotenbeschriftungen wurden zufällig aus einem Wörterbuch ausgewählt. Bei jedem Versuch wurde ein Datenbaum zufällig ausgewählt und modifiziert, so daß er als Query benutzt werden konnte. Es wurden jeweils 10 gleiche Versuche durchgeführt, von denen das Mittel berechnet und in die Graphiken gezeichnet wurde.

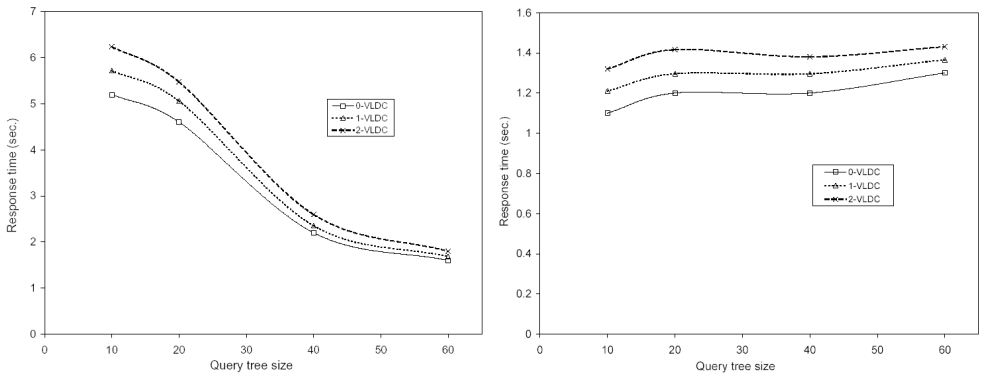

Abbildung 6: Laufzeiten in Abhängigkeit zur Querygröße

In Abbildung 6 sieht man die Laufzeiten von „ATreeGrep“ mit ein, zwei und drei Wildcards (VLDC) in der Query, in Abhängigkeit von der Querygröße. In der linken Graphik wurde ein Wörterbuch der Größe 50, rechts der Größe 1000 benutzt. Man sieht sehr schön, wie die Laufzeit mit der Anzahl der Wildcards skaliert. Außerdem erkennt man, daß die Laufzeit mit wachsender Querygröße abnimmt, wenn das Wörterbuch klein ist, und nahezu konstant bleibt bei großem Wörterbuch. Dies erklärt sich aus dem Filtern mit Hilfe der Vater-Kind-Paare, die bei großen Queries ein sehr gutes Filterkriterium sind. Dagegen hilft das Filtern bei kleinen Queries und kleinem Wörterbuch nicht viel, da dann viele mögliche Datenbäume gefunden werden, deren Überprüfung mit „Pathfix“ viel Zeit kostet.
In Abbildung 7 wird „Pathfix“, ohne Filtern, mit „ATreeGrep“, mit Filtern, verglichen. Man erkennt, daß „Pathfix“ wie erwartet etwa lineare Laufzeit in Abhängigkeit

der Querygröße hat, wohingegen „ATreeGrep“ dank des Filterns fast eine konstante Laufzeit aufweist.

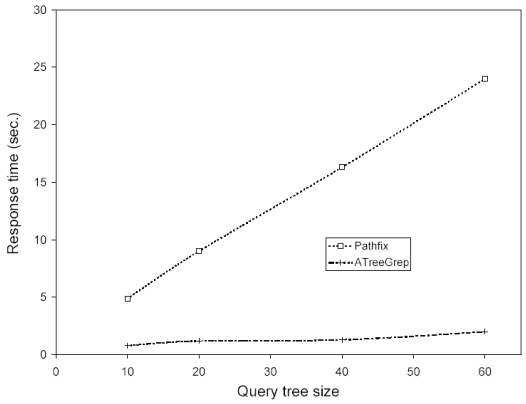

Abbildung 7: Gegenüberstellung von Pathfix und ATreeGrep

2.8 Fazit

Vorgestellt wurde ein neuer Ansatz zur Lösung des Approximate Nearest Neighbor Search Problems für ungeordnete Bäume. Durch die Einführung eines neuen Distanzmaßes, das nicht auf Tree Editing Distanz beruht, war es möglich, einen praktikablen Algorithmus mit polynomieller Laufzeit zu entwerfen, was eine deutliche Verbesserung zur Tree Editing Distanz auf ungeordneten Bäumen ist, da deren Berechnung ein NP-vollständiges Problem darstellt. „ATreeGrep“ wurde jedoch nicht mit anderen bestehenden Algorithmen zur Ähnlichkeitssuche auf XML-Daten verglichen, jedoch sind zwei Applikationen für jeden Interessierten im Internet verfügbar unter
`http://www.njit.edu/mediadb/xmlsearch/main.htm/`
`http://www.treebase.org/treebase/console.html`

Die Autoren des Papers haben für die Zukunft in Aussicht gestellt, daß sie den hier vorgestellten Algorithmus für geordnete Bäume anpassen möchten und andere Typen wissenschaftlicher Datenbanken.

3 Ähnlichkeitssuche mittels Relaxationen in Join-Plänen

3.1 Einleitung

[AYCS02] befassen sich mit der Ähnlichkeitssuche auf ungeordneten Bäumen, insbesondere in XML-Daten. Sie führen dieses Problem auf die Evaluierung eines Join-Plans zurück, indem sie die Suchanfrage mittels *Relaxationen* verallgemeinern und diese in einem Join-Plan ausdrücken. Die Autoren implementieren Optimierungsstrategien, mittels welcher die Effizienzverluste bei der Evaluierung solcher verallgemeinerten Join-Pläne gering gehalten werden können.

Wir werden im folgenden mit der Darstellung des Suchproblems beginnen. Anschließend werden wir zeigen, wie eine exakte Anfrage als Join-Plan formuliert wird,

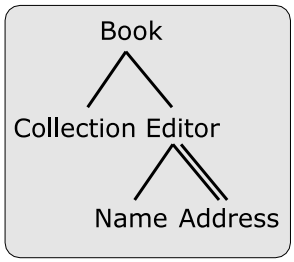

Abbildung 8: Ein Beispielanfragemuster mit den drei möglichen Musterelementen: benannten Knoten, Mutter-Kind-Beziehungen und eine Vorfahre-Nachfahre-Beziehung (doppelt gezeichnete Kante).

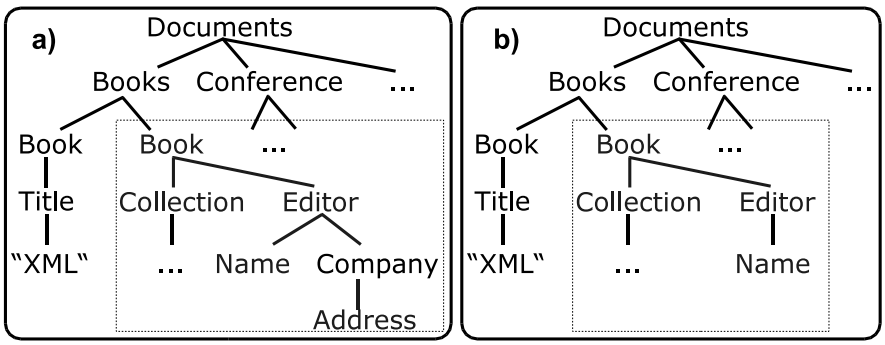

Abbildung 9: a) Ein exaktes Vorkommen des Beispielmusters aus Abb. 8 in einem angenommenen, als Baum dargestellten XML-Dokument. b) Ein ähnliches Vorkommen desselben Musters. Hier fehlt der geforderte Nachfahre „Address“.

und werden dann die Umsetzung approximativer Anfragen mittels Relaxationen in einen Join-Plan darlegen. Wir stellen danach die Algorithmen von [AYCS02] zur Evaluierung solcher Join-Pläne vor und umreißen kurz die experimentellen Ergebnisse.

3.2 Problemstellung

Die Benutzerin gibt ein Baummuster als Anfrage[1] vor und möchte exakte und/oder ähnliche Vorkommen aus gegebenen XML-Daten zurückbekommen. Als Elemente einer solchen Anfrage können benannte Knoten, Mutter-Kind-Beziehungen und Vorfahre-Nachfahre-Beziehungen benutzt werden. Abbildung 8 zeigt ein Beispiel eines Anfragemusters. Für einen exakten Treffer müssen die Knoten des Musters auf entsprechend benannte Knoten eines XML-Dokuments sowie analog die Mutter-Kind-Beziehungen auf einfache Kanten zwischen den Knoten abgebildet werden, und Vorfahre-Nachfahre-Beziehungen können auf beliebig lange Pfade zwischen den angegebenen Knoten abgebildet werden. Da die XML-Daten und die Anfrage als ungeordnete Baumdaten modelliert werden, werden Ordnungsverhältnisse zwischen Geschwisterknoten vernachlässigt. Abbildung 9 illustriert einen exakten und einen ähnlichen Treffer für das Beispielmuster.

Die Autoren behandeln zwei Arten von Suchanfrage: Erstens die *Schwellenwertanfrage*, bei der die Benutzerin einen numerischen Ähnlichkeitswert spezifiziert und alle Treffer erwartet, die diesen Wert erreichen. Zweitens die *Beste-k-Suche*, bei

[1] Die Begriffe „Anfrage“ und „(Baum)Muster“ sind in den folgenden Ausführungen als synonym zu verstehen.

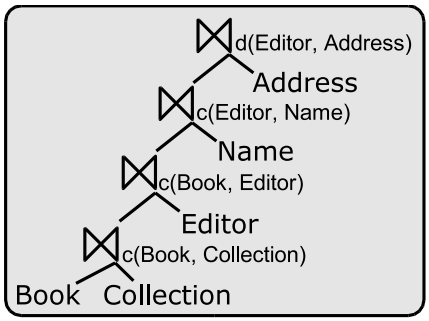

Abbildung 10: Ein linkstiefer Join-Plan für das Anfragemuster aus Abb. 8.

der die Benutzerin nach den k ähnlichsten Treffern sucht, unabhängig von deren tatsächlicher Ähnlichkeit.

3.3 Suchmuster als Join-Plan

Zur Umsetzung der approximativen Suche wird ein Join-Algorithmus als eine „black box“ benutzt. Die Autoren benötigen lediglich zwei allgemeine Voraussetzungen für den verwendeten Algorithmus:

- Die XML-Daten, auf denen gesucht werden soll, liegen für die Anwendung des Join-Algorithmus adäquat aufgearbeitet vor, im Allgemeinen in Form irgendwelcher Indexstrukturen.
- Der Join-Algorithmus erlaubt es, auf den Daten mittels zweier Prädikate Joins auszuführen, die ausdrücken sollen, daß zwei Knoten in der Mutter-Kind-Beziehung (das wird mit `c(.,.)` bezeichnet) bzw. in der Vorfahre-Nachfahre-Beziehung stehen (dieses wird mit `d(.,.)` bezeichnet). Beispielsweise würde ein Join mit der Relation `c(Book, Editor)` die Menge aller Knotenpaare `Book` und `Editor` liefern, für die gilt, daß der `Editor` ein Kind des `Book`-Knotens ist.

Mittels der beiden Prädikate `c(.,.)` und `d(.,.)` kann ein Suchmuster direkt in einen Join-Plan übersetzt werden, indem jede Kante als Join mit dem ihr entsprechenden Prädikat umgesetzt wird. Man geht iterativ die Kanten durch und erweitert je Kante den entstehenden Join-Plan um einen neuen Join, der die bisherige Ergebnismenge durch eine neue Bedingung mit einer der beiden Relationen einschränkt. Man beachte, daß damit das Vorgehen nicht eindeutig festgelegt wird, z.B. die Reihenfolge der Kantenumwandlung nicht vorgeschrieben wird. Die Autoren lassen solche und ähnliche Details, insbesondere Fragen der Optimierung bei der Planerstellung, ausdrücklich außen vor und benutzen in ihren Ausführungen sog. „linkstiefe“ Pläne: Sie gehen das Suchmuster *breadth-first* durch und fügen den Join für jede neue Kante „oben“ an den bisherigen Join-Plan. Abb. 10 zeigt den Join-Plan, der bei diesem Vorgehen für das Beispielmuster aus Abb. 8 entsteht.

Dieser Join-Plan liefert genau die Tupel, die allen Bedingungen des gegebenen Musters genügen, findet also die genauen Vorkommen des Musters. Um ähnliche Treffer zu bekommen, wird der Plan mittels Relaxationen erweitert, die im folgenden Abschnitt vorgestellt werden.

3.4 Relaxationen

Eine Relaxation ist allgemein gesprochen eine strukturelle Veränderung eines Musters M, die M verallgemeinert: Für das relaxierte Muster M^R bekommt man als

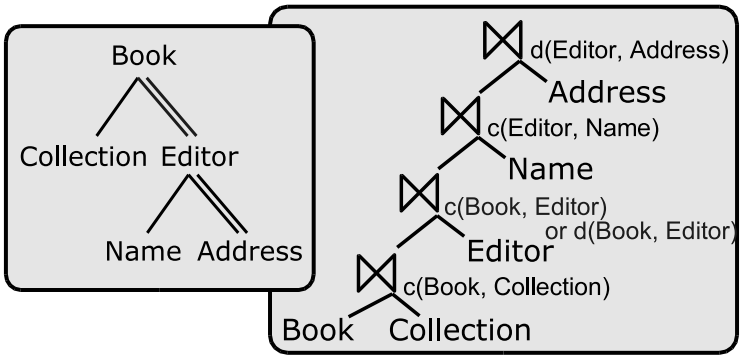

Abbildung 11: Ein Beispiel für eine Kantenrelaxation: Die Mutter-Kind-Kante zwischen „Book“ und „Editor“ aus dem Beispielmuster (Abb. 8) wird in eine Vorfahre-Nachfahre-Kante geändert und im Join-Plan das Prädikat des dazugehörigen Joins erweitert.

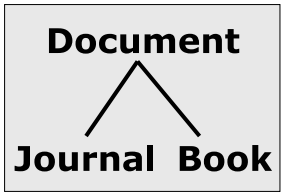

Abbildung 12: Eine Typhierarchie, die „Journal“- und „Book“-Knoten als Untertypen eines gemeinsamen Übertypen „Document“ festlegt.

Treffermenge eine Übermenge der Treffer von M zurück. Die wiederholte Anwendung verschiedener Relaxationen auf ein Muster bewirkt, daß immer allgemeinere Versionen dieses Musters entstehen. [AYCS02] benutzen genau vier Arten von Relaxationen, die sie im Join-Plan für das gegebene Suchmuster umsetzen. Diese Relaxationen und ihre Umsetzung sind im Einzelnen die folgenden:

Kantenverallgemeinerung Bei einer Kantenverallgemeinerung wird eine Mutter-Kind-Vorgabe in eine Vorfahre-Nachfahre-Beziehung umgewandelt; wir lassen also hinterher zwischen den verbundenen Knoten beliebig lange Pfade zu. Im Join-Plan wird diese Relaxation realisiert, indem das Join-Prädikat für die betreffende Kante mit einem logischen `OR` um die Vorfahre-Nachfahre-Beziehung erweitert wird: Wenn die Join-Bedingung vorher `c(Mutter, Kind)` war, lautet sie relaxiert `c(Mutter, Kind) OR d(Mutter, Kind)`.[2] Abbildung 11 zeigt ein Beispiel einer Kantenrelaxation.

Knotenverallgemeinerung Eine Knotenverallgemeinerung verändert eine Knotenbezeichnung mit Hilfe einer Typhierarchie. Eine solche Typhierarchie definiert Sub- und Subertypbeziehungen zwischen Knotenbeziehungen. Sie muß im allgemeinen vorgegeben sein, etwa von einem Experten anhand der XML-Daten definiert worden sein.

Bei einer Knotenverallgemeinerung wird eine Knotenbezeichnung durch einen Supertypen aus der Hierarchie ersetzt. Dieselbe Operation wird zur Umsetzung im Join-Plan verwendet: Jedes Vorkommen des betreffenden Knoten-

[2] Hier wie in folgenden ähnlichen Fällen wird ohne Einschränkung der Allgemeinheit davon ausgegangen, daß das OR kurzschließt, daß also der zweite Term nicht ausgewertet wird, falls die erste Bedingung erfüllt ist.

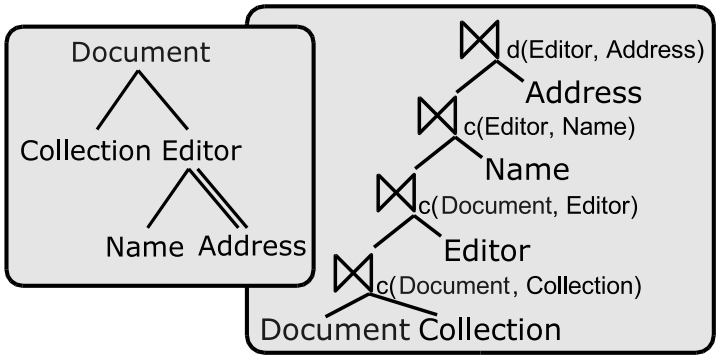

Abbildung 13: Ein Beispiel für eine Knotenrelaxation mittels der Hierarchie aus Abb. 12. Der „Book"-Knoten des Beispielmusters wird verallgemeinert. Seine Vorkommen werden im Join-Plan analog ersetzt.

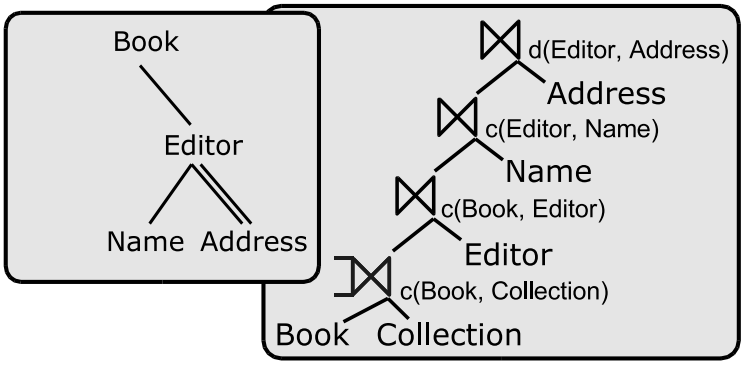

Abbildung 14: Relaxation mittels optionaler Blätter: Der „Collection"-Knoten des Beispielmusters wird aus dem Muster gelöscht. Im Join-Plan wird der entsprechende Join zu einem äußeren Join.

namens wird durch den Supertypen ersetzt. Wo in der ursprünglichen der spezifische Typ verlangt war, ist damit nach der Relaxation jeder Typ erlaubt, der ebenfalls Subtyp des eingesetzten Supertyps ist. Abb. 12 zeigt eine einfache Beispielhierarchie, mit der durch Relaxation zum gemeinsamen Supertypen `Document` alle `Journal`-Knoten als ähnliche Treffer für `Book`-Knoten (und vice versa) zugelassen werden können. Abb. 13 veranschaulicht eine Knotenrelaxation am Beispielmuster aus Abb. 8.

Optionales Blatt Ein Blatt eines Suchmusters wird optional gemacht, indem es aus dem Muster gelöscht wird. Die Umsetzung im Join-Plan verlangt die Einführung eines *linken äußeren Joins*. Die bisher betrachteten Joins waren *innere Joins*, die genau die Tupel als Ergebnismenge liefern, für die die Join-Bedingung erfüllt ist. Beim *linken äußeren Join* werden dagegen *alle linken* Ergebnisstupel oder -knoten behalten, unabhängig davon, ob sie tatsächlich in der vom Join verlangten Relation stehen. Ein äußerer Join mit der Relation `c(Book, Editor)` z.B. würde *alle* `Book`-Knoten des Gegenstandsbereiches zurückliefern, auch solche, die keinen `Editor` als Kind haben. Diese `Book`-Knoten bekämen in der Ergebnismenge für den `Editor` einen Nullwert eingetragen; der `Editor` wäre mithin optional. Abb. 14 veranschaulicht diese Relaxation und ihre Umsetzung am Beispielmuster aus Abb. 8.

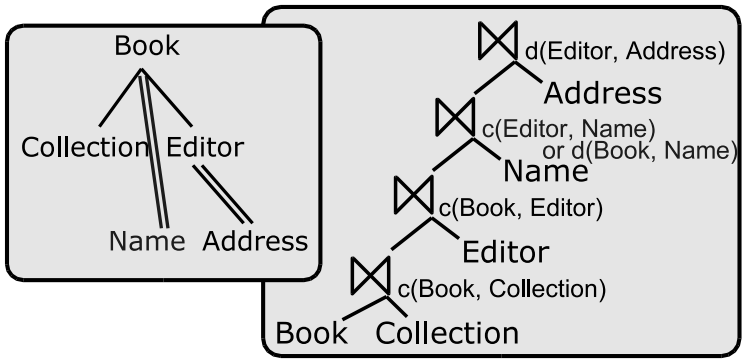

Abbildung 15: Beispiel einer Teilbaumbeförderung: Das einzelne Blatt „Name“ wird mit seiner Großmutter verbunden, und im Join-Plan wird die dazugehörige Relation entsprechend erweitert.

Teilbaumbeförderung Bei der Teilbaumbeförderung wird der Elternknoten eines gewählten Teilbaums „umgangen“, indem der Teilbaum mit seinem Großmutterknoten durch eine Vorfahre-Nachfahre-Kante verbunden wird. Auf diese Weise wird der umgangene Elternknoten optional. Im Join-Plan drückt sich diese Relaxation wie bereits bei der Kantenrelaxation in einer Erweiterung des entsprechenden Join-Prädikates aus: Wo vorher die Kind-Beziehung zum Teilbaum gefordert war, wird nun durch Anfügung eines `OR`-Terms auch zugelassen, daß der Teilbaum stattdessen Nachfahre der Großmutter sein kann. Abb. 15 veranschaulicht diese Relaxation und ihre Umsetzung am Beispielmuster aus Abb. 8.

Am Beispiel der Teilbaumbeförderung wird deutlich, daß die vier Relaxationen durchaus nicht disjunkte Treffermengen erzeugen: Z.B. läßt effektiv die Teilbaumbeförderung alle beliebigen Umbenennungen des umgangenen Mutterknotens zu, die man zum Teil auch mittels einer Knotengeneralisierung (siehe Abschnitt 3.4) erreichen kann. Jede mögliche Knotengeneralisierung kann also auch als Teilbaumbeförderung realisiert werden. Die Autoren thematisieren diese Redundanz jedoch nicht.

Diese vier Relaxationen werden auf jede mögliche Weise auf das gegebene Suchmuster angewendet, wodurch man den allgemeinsten möglichen Join-Plan erhält. Für das Muster aus Abb. 8 ist dieser Plan in Abb. 16 abgebildet.

3.5 Gewichtung und Bewertung

Um die Menge der ähnlichen Treffer gemäß der von der Benutzerin gestellten Anfrage einschränken zu können, müssen die Treffer hinsichtlich ihrer Ähnlichkeit bewertet werden. Dazu muß die Benutzerin ihr Anfragemuster gewichten. Sie muß für jede Kante und jeden Knoten des Musters zwei Gewichte angeben, ein exaktes Gewicht und ein kleineres, sog. relaxiertes Gewicht. Daraus bestimmt sich die Bewertung jedes Elements: Wird das Element in einem Vorkommen exakt wiedergefunden, bekommt es als Wert das exakte Gewicht. Kommt es jedoch nur relaxiert vor – wird etwa für eine Kante statt einer verlangten Mutter-Kind-Beziehung eine Vorfahre-Nachfahre-Beziehung gefunden –, bekommt es einen Wert zwischen dem exakten und dem relaxierten Gewicht. Denkbar ist an der Stelle jede beliebige monotone Funktion, die die Bewertung z.B. abhängig machen kann von der Länge eines

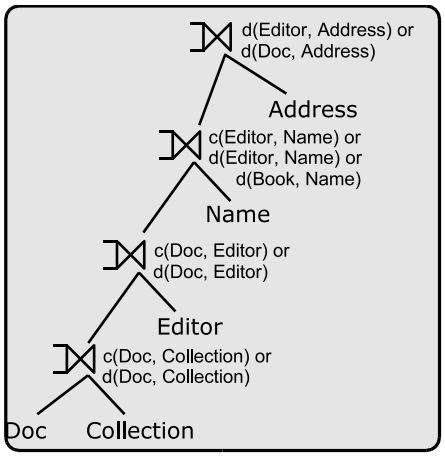

Abbildung 16: Der Join-Plan aus Abb. 10 nach seiner vollständigen Relaxierung.

relaxierten Pfades o.ä.

Die Bewertung eines Vorkommens läßt sich mittels der Bewertungen der einzelnen Elemente als Summe dieser Werte berechnen. Kommt in dem Vorkommen ein Element, z.B. ein optionales Blatt nicht vor, werden dafür auch keine Punkte angerechnet. Ein exaktes Vorkommen des Anfragemusters bekommt nach diesem Vorgehen das mögliche Maximum, nämlich die Summe aller exakten Gewichte zugewiesen.

3.6 Evaluierung des Join-Plans

Der naive Ansatz, eine approximate Suche auszuführen, das sogenannte *post-pruning*, besteht in folgendem Vorgehen: Das Suchmuster wird als Join-Plan umgesetzt, dieser wird vollständig relaxiert und die Ergebnistreffer werden anhand der gegebenen Gewichte bewertet. Dann werden alle Ergebnisse weggeworfen, die nicht das von der Benutzerin spezifizierte Kriterium erreichen.

Der relaxierte Join-Plan in Abb. 16 verdeutlicht die Schwäche dieses Ansatzes: Bis auf den Wurzelknoten `Book` sind alle Knoten optional (da alle Joins äußere Joins sind), und alle Join-Prädikate sind mehrteilige `OR`-Formeln. Es ist folglich zu erwarten, daß bei bei der Evaluierung dieses Plans große Datenmengen zu verarbeiten sind und zugleich viele nur sehr entfernt ähnliche Ergebnisse geliefert werden, die anschließend wieder gelöscht würden. Die Algorithmen der Autoren zielen daher darauf ab, schon während der Evaluierung des Join-Plans die Menge der Ergebnisse gezielt auf zwei Weisen zu verringern:

- Es können frühzeitig Vorkommen identifiziert und gelöscht werden, von denen sicher ist, daß sie nicht das gegebene Suchkriterium erreichen können. Dies leisten zwei Algorithmen, `thres` für die Schwellenwertsuche und `top-k` für die Beste-k-Suche.
- Es können frühzeitig Relaxationen erkannt und zurückgenommen werden, die nur irrelevante Vorkommen erzeugen werden. Dies leistet eine Erweiterung von `thres` namens `opti-thres` für die Schwellenwertsuche. Eine entsprechende Umsetzung für die Beste-k-Suche wird nicht vorgestellt.

Das Prinzip ist bei beiden Vorgehen dasselbe: Mittels zusätzlicher, im Voraus berechneter Gewichte wird während der Evaluierung jedes Knotens im Join-Plan bestimmt, welche Ergebnisse bzw. welche Relaxationen das spezifizierte Kriterium

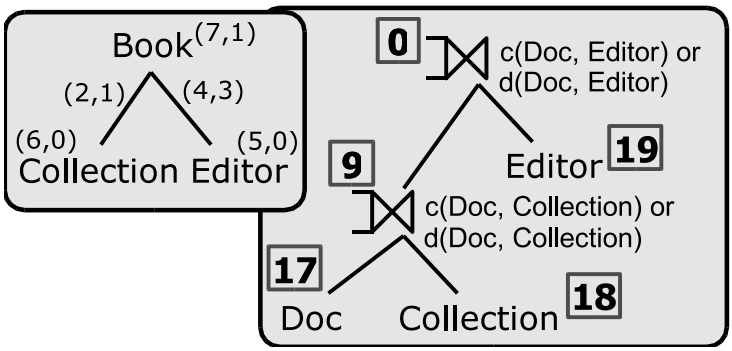

Abbildung 17: Ein gewichtetes Anfragemuster (je Element mit exaktem und relaxiertem Gewicht) und die von `thres` dazu berechneten `maxW`-Werte im relaxierten Join-Plan.

nicht erreichen können, und diese werden gelöscht bzw. zurückgenommen. Wir werden zunächst zeigen, wie dies bei `thres` geschieht, und anschließend `opti-thres` und `top-k` vorstellen.

3.6.1 Thres

`thres` wird benutzt, um eine Schwellenwertsanfrage zu beantworten. Der Algorithmus löscht während der Evaluierung der einzelnen Knoten des Join-Plans Zwischenergebnisse, die garantiert nicht mehr die von der Benutzerin vorgebenene Schwelle erreichen können. Dazu wird vor der Evaluierung des Join-Plans für jeden Knoten im Plan ein Wert `maxw` berechnet, der angibt, wieviele Punkte ein Treffer *nach* Evaluierung des jeweiligen Knotens maximal noch bekommen kann. Abb. 17 illustriert das Prinzip an einem kleinen Anfragemuster: Bei der Evaluierung eines Blattes im Join-Plan kann man für alle verbleibenden Knoten des Planes noch volle Punktzahl bekommen; bei jedem Join hingegen nur noch für die hiernach auszuwertenden Elemente, d.h. die über dem jeweiligen Join verbleibenden Elemente.

Hat man diese Gewichte berechnet, können während der Evaluierung des Join-Plans nach der Ausführung jedes Joins die erhaltenen Teilergebnisse daraufhin überprüft werden, ob die Summe der Bewertung des jeweiligen Teilergebnisses und `maxW` noch die gegebene Schwelle erreicht. Teilergebnisse, für die das nicht der Fall ist, können den Schwellenwert auch später nicht mehr erreichen und werden gelöscht, wodurch sie bei späteren Joins nicht mehr berücksichtigt werden.

3.6.2 Opti-Thres

`opti-thres` ergänzt `thres` um die weitere Optimierung, Relaxationen zurückzunehmen und damit effizientere Join-Operationen auszuführen, falls die Relaxation nur irrelevante Ergebnisse erzeugen kann. Die drei möglichen Relaxationen, die je Join zurückgenommen werden können, sind

- die Rückumwandlung des äußeren Joins zu einem inneren Joins,
- die Rücknahme einer eventuellen Generalisierung des rechten Kindknotens des Joins (z.B. von `Document` zurück zu `Book`) und
- die Löschung alternativer `OR`-Terme in der Join-Relation.

Dazu berechnet `opti-thres` vor der Evaluierung des Plans je Join-Knoten drei zusätzliche Gewichte, die angeben, wieviele Punkte ein Vorkommen nach dem jeweiligen Knoten *unter Annahme der jeweiligen Relaxation* noch bekommen kann. Während der Evaluierung merkt sich `opti-thres` ferner die Bewertung `b` des jeweils besten Teilergebnisses. Vor Ausführung jedes Joins kann nun bestimmt werden, ob die Summe aus `b` und den jeweiligen Gewichten noch das gegebene Schwellenkriterium erfüllt. Falls dies für ein Gewicht nicht der Fall ist, wären alle durch die entsprechende Relaxation erzeugten Ergebnisse irrevelant, und die Relaxation wird zurürckgenommen, wodurch kompaktere Joins entstehen.

Auf die Menge der Teilergebnisse jedes dieser Joins wird anschließend `thres` angewendet.

3.6.3 Top-K

Bei `top-k` handelt es sich um eine Variation von `thres` für die Beste-k-Anfrage. Während `thres` mit einer vorgebenen, festen Schwelle arbeitet, verwendet `top-k` eine dynamische Schwelle: Der Algorithmus merkt sich während der Evaluierung das Ergebnis `kW` des k-ten Ergebnisses. Mit dieser Schwelle kann dieselbe Überprüfung wie bei `thres` vorgenommen werden: Wenn die Bewertung eines Teilergebnisses plus dem `maxW` des Join-Knotens nicht den Wert `kW` erreicht, wird das Ergebnis auf jeden Fall hinter dem jetzigen k-ten Ergebnis eingeordnet werden müssen und kann gelöscht werden.

Man beachte, daß dieses Vorgehen wesentlich davon abhängt, daß die Ergebnismenge über die äußeren Joins immer größer wird, wir also keine inneren Joins verwenden dürfen, die die Ergebnismenge wieder einschränken können, wodurch evtl. vorher gelöschte Ergebnisse doch unter die besten k hätten kommen können. Daher wird auch der in `opti-thres` verwendete Ansatz nicht von den Autoren für die Beste-k-Anfrage übertragen.[3]

Ein Problem, das die Autoren nicht erwähnen, besteht im nicht deterministischen „Zerschneiden“ von Mengen gleichbewerteter Treffer durch den Algorithmus: Da die Bewertungen der Treffer die Summen der Gewichte sind und es somit möglicherweise viele Treffer gibt, die dieselbe Bewertung bekommen, wird es sehr wahrscheinlich, daß die Grenze der besten k inmitten eine solche Menge gleichbewerteter Treffer fällt. Welche dieser eigentlich gleich guten Treffer dann genommen und welche weggeworfen werden, ist nicht genauer spezifiziert und hängt z.B. von der Reihenfolge der Joins ab.

3.7 Experimentelle Ergebnisse

Die Autoren haben ihre Algorithmen exemplarisch mittels drei kleiner Anfragen auf der Trierer DBLP getestet, um die Vorteile gegenüber den naiven *post-pruning* zu demonstrieren. Da die Ergebnisse für die drei Verfahren ähnlich sind, sollen hier nur die Ergebnisse für `thres` vorgestellt werden. Abb. 18 zeigt die erzeugten Datenmengen und die kumulative Zeit für eine Anfrage aus drei Joins für Läufe von `thres` mit drei verschiedenen Schwellenwerten gegenüber dem naiven *post-pruning* Ansatz. Die Schwelle von acht entspricht dabei der Forderung nach exakten Treffern für das Muster. Besonders in der Darstellung der Datenmenge zeigt sich deutlich, wie `thres` mit steigender Schwelle immer mehr Teilergebnisse immer früher löscht. Man beachte jedoch, daß hinsichtlich der erzeugten Datenmengen die Läufe von `thres` mit den Schwellen von drei und fünf im Endeffekt relativ dicht beieinanderliegen (ca. 134000 bzw. 87000 Datensätze), im Zeitdiagramm jedoch deutlich divergieren (ca.

[3] Die Frage, inwieweit es möglich wäre, den Ansatz von `opti-thres` nur partiell, nämlich für die zwei anderen der drei bei `opti-thres` betrachteten Relaxationen hier umzusetzen, wird von den Autoren leider nicht erwogen.

18 Sekunden gegenüber sieben Sekunden). Das könnte darauf hindeuten, daß die Reihenfolge der Joins, also die Optimierung des Join-Plans von großer Bedeutung für den Zeitgewinn ist.

3.8 Zusammenfassung und Diskussion

Die Autoren zeigen durch die Anwendung von Relaxationen einen intuitiven Weg auf, um approximate Suche auf XML-Daten in Join-Architekturen umzusetzen, und sie zeigen, wie die Evaluierung solcher Joins nachhaltig optimiert werden kann. Da wenig Voraussetzungen an die Join-Architektur gestellt werden, kann der Ansatz relativ einfach als Erweiterung für entsprechende Systeme benutzt werden. Ferner können in der Knotengeneralisierung sogar Ontologien sinnvoll integriert werden, sofern diese vorhanden sind.

Möglicherweise problematisch ist die Effizienz des Ansatzes bei größeren Anfragen, Typhierarchien oder anderen Parametern. Die Autoren behandeln die Frage nach der Effizienz nur außerordentlich peripher, aber wie bei der Vorstellung der experimentellen Ergebnisse im vorigen Abschnitt angemerkt, ist es denkbar, daß Details wie die Optimierung des Join-Plans von bedeutendem Einfluß für die Effizienz sein können. Das gilt erst recht, wenn wirklich ausgefeilte, z.B. tiefere Typhierarchien verwendet werden sollen als die in den Beispielen benutzten. Solche sind in den Algorithmen nicht vorgesehen, und es ist nicht auf Anhieb ersichtlich, auf welche Weise z.B. in `thres` eine Knotenrelaxierung über verschiedene Tiefen einer Typhierarchie realisiert werden kann. Auch stellt sich dann die von den Autoren außen vor gelassene Frage, wie eine abgestufte Bewertungsfunktion für relaxierte Vorkommen von Kanten und Knoten sich auf die Laufzeiten auswirken würde, da dafür bei jedem relaxierten Vorkommen die Länge des relaxierten Pfades (für Kanten in den XML-Daten, für Knoten in der Typhierarchie) ermittelt werden müßte.

Als Fazit läßt sich damit vielleicht sagen, daß [AYCS02] einen hinsichtlich seiner Eleganz vielversprechenden Ansatz vorstellen, der in einigen technischen Details durchaus noch verbesserungswürdig ist.

4 Unscharfe Joins über XML-Daten

Im Gegensatz zu den beiden oberen Abschnitten, wo wir eine (eher kleine) Anfrage und ein (eher größeres) XML Dokument haben, werden wir hier ein paar join-Algorithmen über beliebige Dokumente kennenlernen.

4.1 Einführung und Anregung

Wie wir an dem Bild 19 sehen können, können oft mehrere XML-Dokumente dieselbe Information beinhalten, aber verschiedene Struturen haben. Deswegen brauchen wir effiziente Techniken, um XML-Dokumente 'approximativ' vergleichen zu können. Jedes mal, wenn man zwei Objekte 'approximativ' vergleichen will, muß man eine Distanzmetrik zwischen den Objekten definieren, um den Vergleich quantitativ bestimmen zu können. Diese Metrik sollte effizient und allgemein sein, d.h. sie sollte auf beliebige Dokumente anwendbar sein.
In diesem Teil werden wir XML-Dokumente als geordnete Bäume ansehen. Deswegen müssen wir effiziente Methoden definieren, um geordnete Bäume 'approximativ' vergleichen zu können.
Im nächsten Abschnitt werden wir eine solche Metrik kennenlernen, und später sehen, wie wir sie optimieren können.

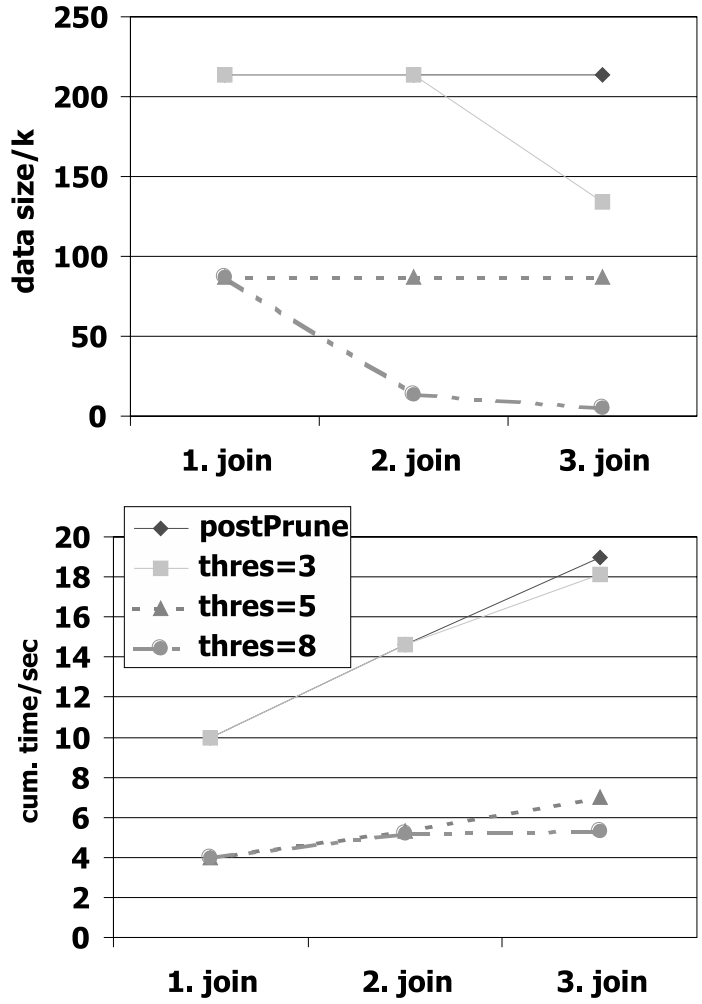

Abbildung 18: `thres` mit drei verschiedenen Schwellenwerten und *post-pruning* im Vergleich. Dargestellt sind die kumulative Zeit bzw. die erzeugte Datenmenge über die drei Joins eines Anfragemusters.

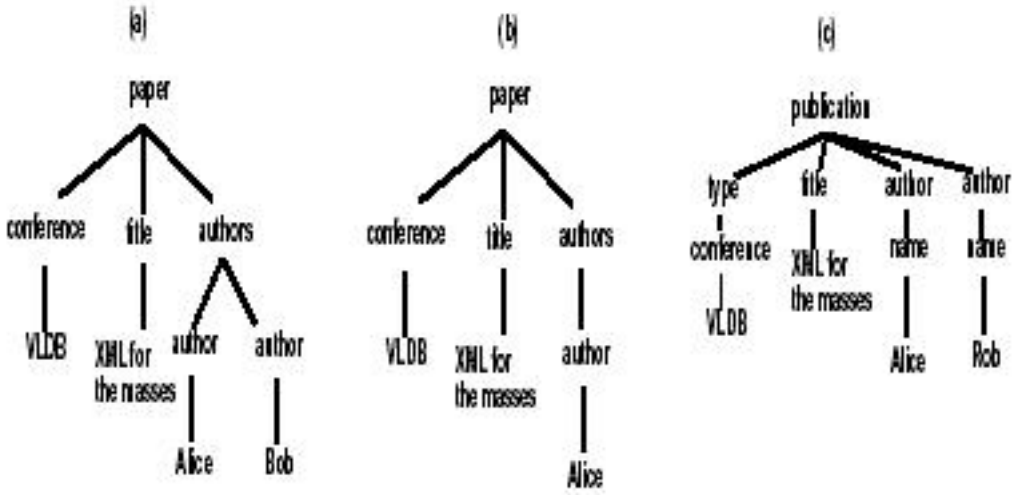

Abbildung 19: Beispiel von XML-Dokumenten

4.2 Editierdistanz

4.2.1 Editierdistanz für Strings

Die Editierdistanz für Strings ist die minimale Anzahl der Editieroperationen (Einfügen, Löschen und Ersetzen) auf einzelnen Zeichen, die man braucht, um ein String in ein anderes umzuwandeln.
Im weiteren wird diese Editierdistanz $ed()$ genannt. $ed()$ kann für Strings der Länge $O(n)$ in $O(n^2)$ berechnet werden.

4.2.2 Editierdistanz für Bäume

Die Editierdistanz für Bäume ist praktisch eine Erweiterung der Editierdistanz für Strings. Also ist die Editierdistanz für zwei gegebene Bäume T_1 und T_2 die minimale Kostsequenz der Baumeditieroperationen (Einfügen, Löschen und Umbenennen) auf einzelnen Knoten, die man braucht, um einen Baum in einen anderen umzuwandeln.
Im weiteren wird die Editierdistanz für Bäume $TDIST()$ genannt. Es gibt einen Algorithmus, der $TDIST()$ in $O(n^4)$ für Bäume der Größe $O(n)$ berechnet. Dieser Algorithmus berechnet ein Mapping zwischen den Knoten beider Bäume. Das Ziel ist es, das Mapping mit den minimalen Kosten zu berechnen. Das berechnete Mapping muß die drei folgenden Bedingungen erfüllen :

1. Ein Knoten darf nicht mit zwei verschiedenen Knoten des anderen Baumes gemappt werden.

2. In dem Mapping muß die Sibling-Ordnung bewahrt werden.

3. Auch die Ancestor-Ordnung muß in dem Mapping bewahrt werden.

Alle Knoten, die nicht gemappt wurden, müssen zum Einfügen oder zum Löschen in Betracht gezogen werden. Diejenigen, die gemappt wurden, müssen je nach Bedarf umbenannt werden.

Als beispiel können wir die Abbildung 20 betrachten. Da haben wir zwei Bäume (a) und (b) und das vom Algorithmus berechneten Mapping. Und wir können sehen, daß der Knoten B in (a) nicht gemappt wurde. Genauso für den Knoten H in (b).

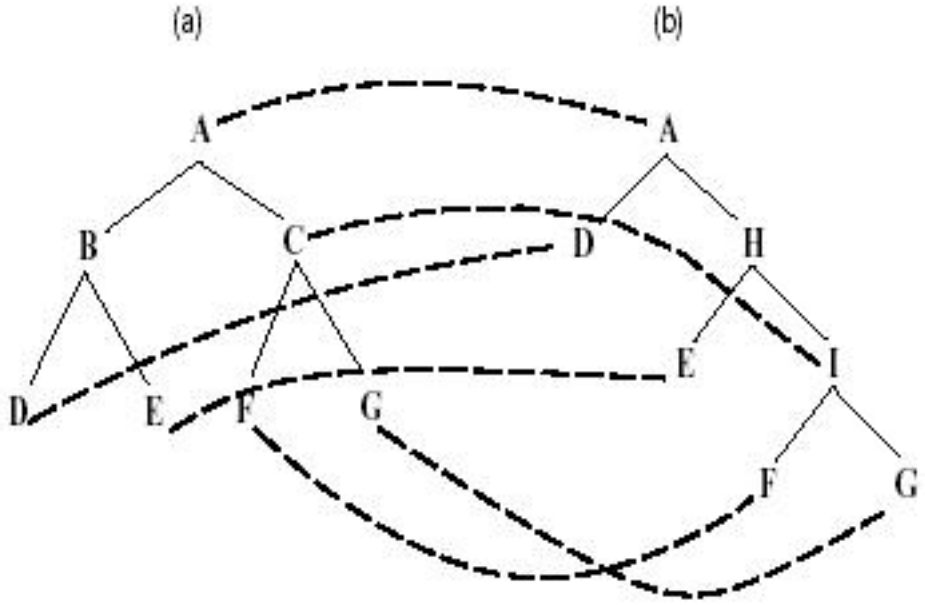

Abbildung 20: Beispiel eines Mappings

Außerdem wurde C mit I gemappt. Um also den Baum (a) in Baum (b) umzuwandeln, muß der Knoten B gelöscht, der Knoten H eingefügt und der Knoten C nach I umbenannt werden. Damit haben wir $TDIST((a),(b)) = 3$, da drei Operationen durchgeführt wurden.

4.3 Problemstellung

Da wir jetzt wissen, wie wir zwei Bäume 'approximativ' vergleichen können, wenden wir uns dem ursprünglich zu lösenden Problem zu.
Seien zwei Datenquellen S_1 und S_2 und ein Schwellenwert τ. Der 'approximative' Join zwischen S_1 und S_2 gibt alle Dokumentenpaare (d_1, d_2) mit $d_1 \in S_1$ und $d_2 \in S_2$ zurück, für die gilt : $TDIST(d_1, d_2) \leq \tau$.
Mehrere Varianten dieses Problems sind möglich. Zum Beispiel könnte man verlangen, daß sich die Distanz zwischen den Dokumenten zwischen zwei gegebenen Schwellwerte befindet (d.h. $\tau_1 \leq TDIST(d_1, d_2) \leq \tau_2$).

Da wir in dem vorherigen Abschnitt $TDIST()$ näher kennengelernt haben, haben wir nun einen (naiven) Algorithmus. Aber wie wir gesehen haben, ist es ziemlich teuer, die Distanz zwischen zwei Dokumente zu berechnen. Deswegen werden wir in den folgenden Abschnitten versuchen, dies etwas effizienter zu machen.

4.4 Exakte Schranken für die Editierdistanz

4.4.1 Untere Schranke

Seien T_1 und T_2 zwei geordnete Bäume. Wir werden jetzt eine untere Schranke für die Distanz zwischen beiden Bäume berechnen. Dafür brauchen wir folgende Erkentnisse.
Sei T ein Baum. $pre(T)$ repräsentiert der Preorder Traversal von T und $post(T)$ der Postorder Traversal.
Wenn es ein Unterschied zwischen den Preorder oder Postorder Traversal von T_1 und T_2 gibt, dann muß es ein Unterschied zwischen beiden Bäume geben. Also :

$pre(T_1) \neq pre(T_2)$ *oder* $post(T_1) \neq post(T_2) \Longrightarrow T_1 \neq T_2$.
Wenn die Distanz zwischen T_1 und T_2 k ist, dann ist das Maximum zwischen den string Editierdistanzen beider Preorder oder Postorder Traversal höchstens k. Mathematisch ausgedrückt heißt es :

$$max\{ed(pre(T_1), pre(T_2)), ed(post(T_1), post(T_2))\} \leq TDIST(T_1, T_2)$$

Mit $LBDIST(T_1, T_2) = max\{ed(pre(T_1), pre(T_2)), ed(post(T_1), post(T_2))\}$ haben wir nun eine untere Schranke für $TDIST()$ hergeleitet, die in $O(n^2)$ für Bäume der Größe $O(n)$ berechnet werden kann, weil hier nur die String-Editierdistanz zum Einsatz kommt.

Hier betrachten wir wieder die beiden Bäume (a) und (b) der Abbildung 20 als Beispiel. Es gilt: $pre((a)) = ABDECFG$ und $pre((b)) = ADHEIFG$. Und damit haben wir $ed(pre((a)), pre((b))) = 3$. Genauso haben wir $post((a)) = DEBFGCA$, $post((b)) = DEFGIHA$ und $ed(post((a)), post((b))) = 3$. Nun gilt:

$$\begin{aligned} LBDIST((a),(b)) &= max\{ed(pre((a)), pre((b))), ed(post((a)), post((b)))\} \\ &= max\{3,3\} \\ &= 3 \\ &\leq TDIST((a),(b)) \end{aligned}$$

4.4.2 Obere Schranke

Die Idee beim Berechnen der oberen Schranke $UBDIST()$ von $TDIST()$ ist, die Anzahl der Möglichkeiten beim Berechnen des Mappings zu reduzieren. Dies wird erreicht, indem man eine zusätzliche Bedingung zu den drei anderen einfügt.
Diese Bedingung ist die folgende : zwei verschiedene Unterbäume des ersten Baumes müssen mit zwei verschiedenen Unterbäumen des zweiten Baumes gemappt werden. Mit dieser Bedingung gibt es einen Algorithmus, der $UBDIST()$ in $O(n^2)$ für Bäume der Größe $O(n)$ berechnen kann.

In Abbildung 21 sehen wir nochmal die beiden Bäume (a) und (b). Das Mapping für $TDIST()$ ist mit gestrichelten Linien gezeichnet (siehe Abschnitt 4.2.2 und Abb. 20). Das berechnete Mapping für $UBDIST()$ ist mit durchgezogenen Linien gezeichnet. Und da sehen wir, daß zusätzlich beide Knoten E in beiden Bäume nicht gemappt wurden (in dem Mapping für $UBDIST()$). Um also Baum (a) in Baum (b) umzuwandeln, müssen die Knoten B und E gelöscht, H und E eingefügt und C nach I umbennant werden. Damit haben wir also: $UBDIST((a),(b)) = 5 \geq TDIST((a),(b))$

4.5 Geschätzte Schranken

Hier werden wir einen anderen Ansatz kennenlernen, um die Kosten des naiven Join-Algorithmus zu reduzieren. Das Ziel ist, die Anzahl der Aufrufe von $TDIST()$ zu reduzieren. Dafür brauchen wir eine *Referenzmenge*.
Sei $K \subset S_1 \cup S_2$ die Referenzmenge, und $k_1, \ldots, k_{|K|}$ eine beliebige Ordnung der Dokumente von K.
Für jedes Dokument $d_i \in S_1$ (bzw. $d_j \in S_2$) berechnen wir den Vektor v_i (v_j) der Länge $|K|$ mit den Einträgen $v_{il} = TDIST(d_i, k_l)$ für alle $1 \leq l \leq |K|$. Genauso haben wir auch $v_{jl} = TDIST(d_j, k_l)$. Wir sollten hier bemerken, daß wir jede andere Metrik zur Berechnung der Vektoren hätten verwenden können (nicht nur $TDIST()$). Da $TDIST()$ eine Metrik ist, gilt die folgende Dreiecksungleichung :

$$\forall 1 \leq l \leq |K|,\ |v_{il} - v_{jl}| \leq TDIST(d_i, d_j) \leq v_{il} + v_{jl}$$

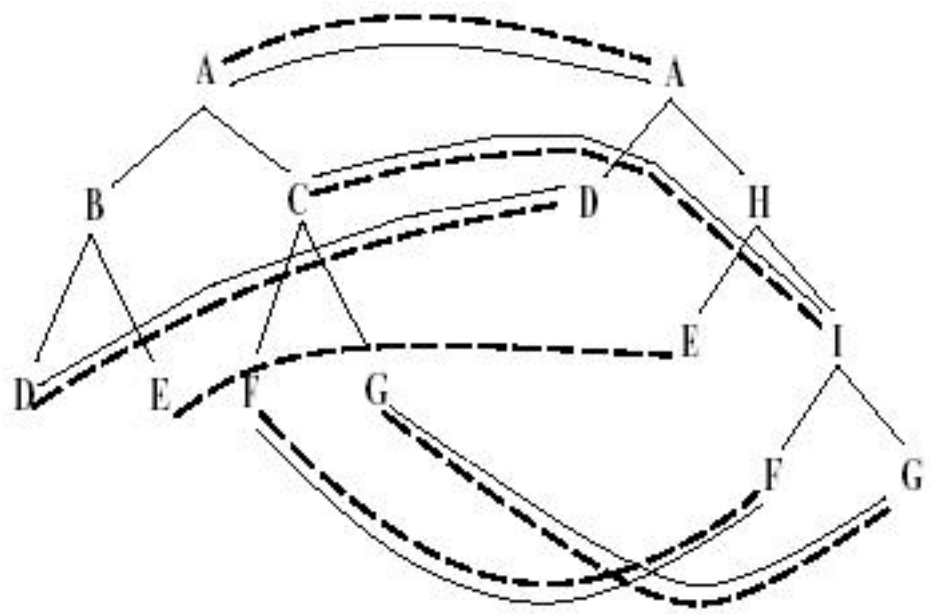

Abbildung 21: Beispiel eines Mappings für die obere Schranke

Damit haben wir folgende 'geschätzte' Schranken für $TDIST(d_i, d_j)$:

- $u_{ij} = min_{l,1 \leq l \leq |K|} v_{il} + v_{jl}$ als obere Schranke und
- $l_{ij} = max_{l,1 \leq l \leq |K|} |v_{il} - v_{jl}|$ als untere Schranke.

Das Problem wäre nun zu wissen, wie wir die Referenzmenge bestimmen können. Dafür werden alle Dokumente in Clustern geteilt, so daß alle Dokumente in einem Cluster sich ziemlich 'ähnlich' sind. Dann wird von jedem Cluster ein Dokument genommen. Genaueres kann man in [GJK+02] nachlesen.

4.6 Algorithmen

Anhand der Schranken, die wir oben berechnet haben, werden wir jetzt ein paar Join-Algorithmen präsentieren. Wir werden hier nur erklären, wie die Algorithmen funktionieren, aber keinen Quellcode geben.

4.6.1 Der naive Algorithmus (N)

Dieser Algorithmus haben schon im Abschnitt 4.3 kennengelernt. Um sagen zu können, ob ein Dokumentenpaar zu dem Ergebnis gehört oder nicht, muß $TDIST()$ aufgerufen werden. Dieser Algorithmus wird im Weiteren N (für naiv) genannt.

4.6.2 Algorithmus mit exakten Schranken (B)

Das ist die erste Verbesserung von N. Hier werden zuerst $LBDIST()$ und $UBDIST()$ aufgerufen, um ermitteln zu können, ob das Dokumentenpaar zu dem Ergebnis gehört oder nicht. Erst wenn wir kein eindeutiges Ergebnis haben, wird $TDIST()$ aufgerufen.
Sei τ der angegebene Schwellwert und (d_1, d_2) das Dokumentenpaar. Wenn $LBDIST(d_1, d_2) > \tau$ gilt, können wir dieses Paar ausschließen, weil auch $TDIST(d_1, d_2) > \tau$ gelten würde. Wenn aber $LBDIST(d_1, d_2) \leq \tau$ gilt, müssen wir $UBDIST()$ aufrufen.

Falls $UBDIST(d_1, d_2) \leq \tau$ ist, dann gehört (d_1, d_2) auf jeden Fall zum Ergebnis. Ansonsten gilt $LBDIST(d_1, d_2) \leq \tau < UBDIST(d_1, d_2)$. Und dann muß $TDIST()$ aufgerufen werden. Wir werden diesen Algorithmus mit B (für *bounds* = Schranken) bezeichnen.

4.6.3 Algorithmus mit geschätzten Schranken (RS)

Dieser Algorithmus funktioniert genauso wie der obere (B). Hier werden nur die im Abschnitt 4.5 berechneten geschätzten Schranken l_{ij} und u_{ij} verwendet, um Paare auszuschließen oder zum Ergenis einzufügen. $TDIST()$ wird dann auch für die restlichen Paare verwendet. Im Weiteren werden wir diesen Algorithmus RS (*reference set*) nennen.

4.6.4 Algorithmus mit beiden Schranken (RSB)

Hier werden zuerst die geschätzten Schranken l_{ij} und u_{ij} benutzt, um Dokumentenpaare auszuschließen oder zum Ergebnis einzufügen. Für die Paare, bei denen wir immer noch keine Gewißheit haben, werden zuerst die exakten Schranken angewendet, bevor $TDIST()$ aufgerufen wird. Wir werden diesen Algorithmus mit RSB (für *reference set, bounds*) bezeichnen.

4.6.5 Algorithmus mit geschätzten Distanzen zur Referenzmenge (RSC)

Wir haben in 4.5 gesehen, daß $TDIST()$ verwendet wird, um die Vektoren zu berechnen. In diesem Algorithmus aber werden die Vektoren mit $LBDIST()$ und $UBDIST()$ berechnet. Dadurch erhalten wir für jedes Dokument d_i (bzw. d_j) zwei Vektoren v_i^l und v_i^u (v_j^l und v_j^l). Und nun gilt dann folgende Dreiecksungleichung:

$$\forall 1 \leq l \leq |K|,\ |v_{il}^l - v_{jl}^u| \leq TDIST(d_i, d_j) \leq v_{il}^u + v_{jl}^u$$

Damit haben wir dann folgende geschätzte Schranken:

- $u_{ij} = min_{l,1 \leq l \leq |K|} v_{il}^u + v_{jl}^u$ als obere Schranke und
- $l_{ij} = max_{l,1 \leq l \leq |K|} |v_{il}^l - v_{jl}^u|$ als untere Schranke.

Dieser Algorithmus wird im Weiteren RSC (für *reference set combined*) bezeichnet.

4.7 Experimentelle Evaluierungen

Alle in 4.6 vorgestellten Algorithmen wurden implementiert. Für die Tests wurden verschiedene Datenmengen benutzt:

- Datenmenge A: Dies ist eine synthetische Datenmenge, die mit dem 'IBM XML data generator' erzeugt wurde und 500 zufällig generierte Dokumente enthält.
- Real DBLP: DBLP Datenbank mit 55MB.

4.7.1 Evaluierung der exakten Schranken

Hier möchten wir wissen, wie gut die in 4.4 berechneten exakten Schranken gegenüber $TDIST()$ sind. Dafür wurde auf der Datenmenge A die paarweisen Distanzen mit $TDIST()$ und auch die Schranken ($LBDIST()$ und $UBDIST()$) berechnet, anschließend die Ratio der Schranken bezüglich $TDIST$. In Abbildung 22 können wir sehen, wieviel Dokumentenpaare sich unter jedem Ratiobereich befinden. Im ersten Bild, das die Ratio von $LBDIST$ repräsentiert, sehen wir, daß es

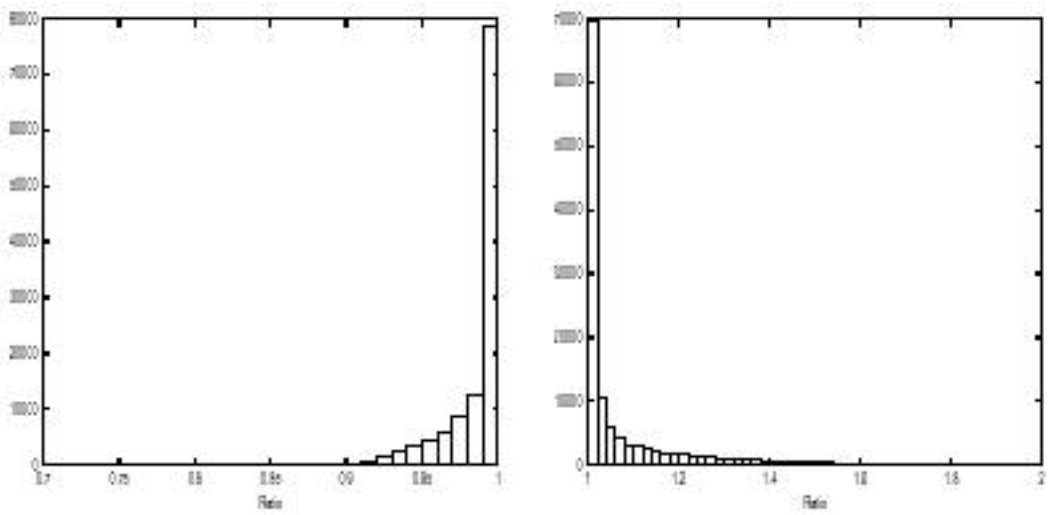

Abbildung 22: Evaluierung der exakten unteren und oberen Schranken

unter 0.9 keine Dokumentenpaare gibt. Genauso sehen wir im zweiten Bild, das die Ratio von $UBDIST$ repräsentiert, daß es nur bis zu der Ratio 1.5 Dokumentenpaare gibt. Aus beiden Bildern folgt, daß die exakten Schranken ziemlich nah an $TDIST$ sind.

4.7.2 Evaluierung der Algorithmen

Hier fragen wir uns, wie gut die Algorithmen, die die Verbesserungen verwenden, im Vergleich zu N sind. Dafür haben wir zwei Fälle zu betrachten.
Im ersten Fall haben wir eine Referenzmenge, die nur fünf Dokumente enthält (Abbildung 23). Da können wir sehen, daß alle Kurven eine glockenartige Form haben. Es liegt daran, daß bei geringe Schwellwerte $LBDIST$ sehr effektiv ist. D.h., mit $LBDIST$ können viele Dokumentenpaare ausgeschlossen werden. Außerdem sehen wir, daß RSC und RSB die schnelleren Algorithmen sind (die sind fast gleich schnell), und daß RS der schlechtere Algorithmus ist, da für die Berechnung der Vektoren $TDIST$ benutzt wird.

Im zweiten Fall enthält die Referenzmenge 100 Dokumente (Abbildung 24). Da sehen wir auch, daß alle Kurven eine glockenartige Form haben. Die Gründe sind dieselben wie im ersten Fall. Hier merken wir aber, daß RSB deutlich schlechter wird, sogar schlechter als B. Es liegt daran, daß die Berechnung der Vektoren deutlich teurer wird.

4.7.3 Evaluierung über DBLP

In dem vorherigen Abschnitt wurden die verschiedenen Algorithmen mit N verglichen. Es wäre aber sehr ineffizient, N auf DBLP laufen zu lassen. Berechnungen zufolge würde es ungefähr 81 Tage dauern. In Abbildung 25 sehen wir, wieviel Zeit mit RSC gebraucht wird mit verschiedenen Schwellwerten.

4.8 Zusammenfassung

In diesem Teil haben wir gesehen, wie wir eine obere und untere Schranke der Baumeditierdistanz für geordnete Bäume berechnen können, und daß sie viel effizienter

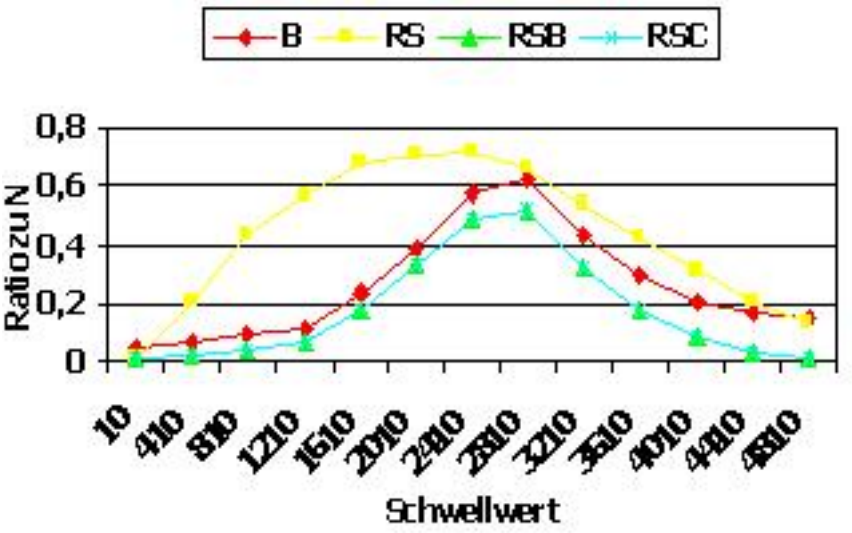

Abbildung 23: Evaluierung der Algorithmen mit einer Referenzmenge der Größe 5

berechnet werden können. Außerdem wurde auch eine generische Methode präsentiert, die auf die Idee der Referenzmenge basiert.
Anhang dieser Optimierungen haben wir dann join-Algorithmen vorgestellt. In allen Algorithmen wurde die Baumeditierdistanz verwendet. Jede andere Metrik, die vielleicht für den Benutzer geeigneter wäre, würde auch funktionieren. Es wäre ziemlich interessant, die hier präsentierten Algorithmen mit anderen Metriken zu testen.

Literatur

[AYCS02] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, March 25–27 2002.

[GJK+02] Sudipto Guha, H.V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, June 3–6 2002.

[SWSZ02] Dennis Shasha, Jason T.L. Wang, Huiyuan Shan, and Kaizhong Zhang. Atreegrep: Approximate searching in unordered trees. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, July 14–16 2002.

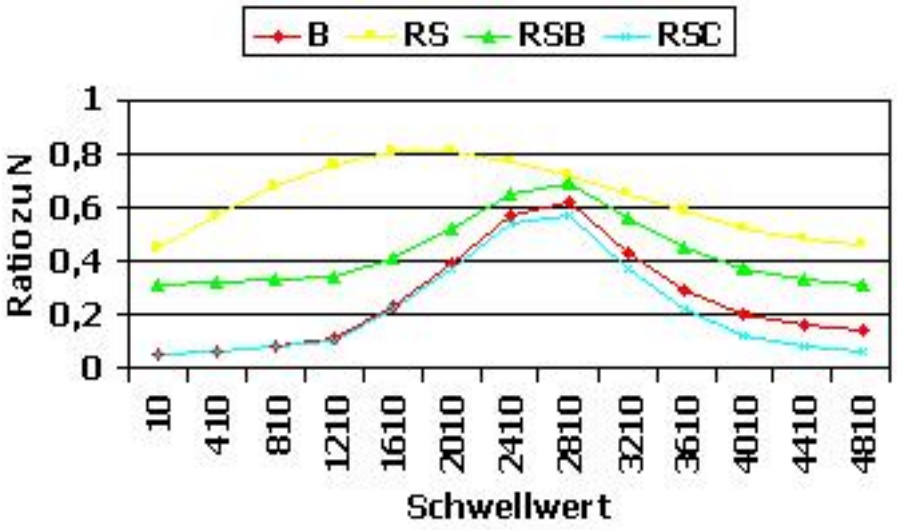

Abbildung 24: Evaluierung der Algorithmen mit einer Referenzmenge der Größe 100

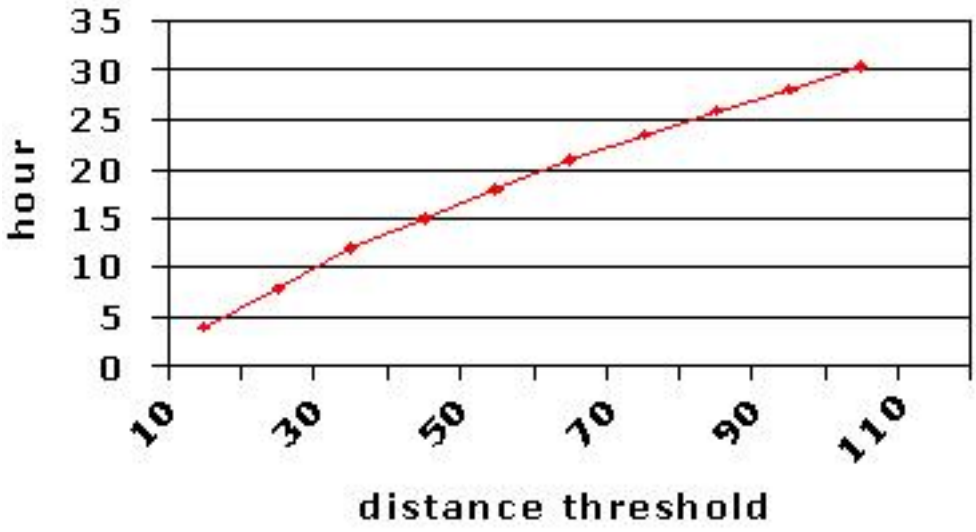

Abbildung 25: Evaluierung über DBLP mit RSC