Universität Konstanz
FB Informatik und Informationswissenschaft
Master-Studiengang Information Engineering

Masterarbeit

Verarbeitung und Visualisierung von XML-Full-Text Daten

zur Erlangung des akademischen Grades eines
Master of Science (M.Sc.)

Studienfach: Information Engineering
Schwerpunkt: *Computer-Science*
Themengebiet: **Informatik der Systeme**

von
Sebastian Gath
(01/556108)

Erstgutachter: Prof. Dr. Marc H. Scholl
Zweitgutachter: Prof. Dr. Daniel A. Keim
Betreuer: Christian Grün (M. Sc.)
Einreichung: 23. April 2009

Konstanzer Online-Publikations-System (KOPS)
URN: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-124664
URL: http://kops.ub.uni-konstanz.de/volltexte/2010/12466/

Zusammenfassung

Diese Master-Arbeit beschreibt die Konzeption, Entwicklung und Implementierung einer Full-Text Erweiterung für ein XML-Datenbanksystem. BaseX, das zu grunde liegende native open-source XML-Datenbanksystem, verfügt über einen effizienten XQuery Prozessor, der im Rahmen dieser Arbeit um die XQuery Full-Text Funktionalität erweitert wird. Neben deren Implementierung wird ein Verfahren zur vollständig automatischen Anbindung von Full-Text Index-Strukturen zur Anfrageverarbeitung eingeführt und die sich hieraus ergebenden weiteren Optimierungen für die Full-Text Anfrageverarbeitung werden aufgezeigt. Zur Darstellung von Volltexten und Suchergebnisse in Volltexten wird ein Visualisierungsverfahren für XML-Full-Text Daten vorgestellt, wobei dessen Implementierung das interaktive Frontend von BaseX erweitert. Die eingeführten Konzepte sind allgemein gültig und auf weitere XML-Datenbanksysteme übertragbar.

Abstract

This Master Thesis illustrates the conception, development and implementation of a Full-Text extension for a XML database system. BaseX, the underlying native open source XML database system, has an efficient XQuery processor, which will be extended by the XQuery Full-Text functionality. In addition to the implementation of the Full-Text functionality, a full-fledged self-acting approach for Full-Text index usage within query processing is presented. The resulting optimizations for Full-Text query evaluation are pointed too. For displaying Full-Text and results of Full-Text searching, a visualization for XML-Full-Text data is described and its implementation extends the BaseX interactive frontend. The introduced concepts are generic and could be assigned to other XML database systems.

Danksagung

Diese Arbeit ist Stefanie Kern gewidmet, die mich zu jedem Zeitpunkt meines Studiums mit Verständnis liebevoll unterstützt hat und mir stets Rückhalt bot. In besonderer Weise möchte ich mich bei meinen Eltern Doris und Dieter Gath bedanken, die mir mit Vertrauen und Respekt das Studium ermöglicht haben.

Besonderer Dank gebührt der Arbeitsgruppe Datenbanken und Informationssysteme, insbesondere Christian Grün und Alexander Holupirek, für die intensive Betreuung und Hilfestellung. Für die Ermöglichung und Begutachtung dieser Arbeit möchte ich Herrn Prof. Dr. Marc H. Scholl und Herrn Prof. Dr. Daniel A. Keim danken.

Bedanken möchte ich mich auch bei meinen Teamkollegen, die mir bei der Bewältigung des Studiums sehr geholfen haben und mir mit ihren individuellen Charaktereigenschaften sehr viel Freude bereiten.

Vielen Dank.

Inhaltsverzeichnis

KAPITEL 1

Motivation und Einleitung in die Problemstellung

1.1 Motivation und Einleitung

Mit der steigenden Verbreitung von XML [5] wachsen auch die Möglichkeiten des Einsatzes von XML-Dokumenten. Sie können zum Datenaustausch zwischen Systemen, als Alternative zur Speicherung hierarchischer Daten oder als Austauschformat für textuelle Daten dienen und eröffnen so ein breites Feld an Textverarbeitungsmöglichkeiten. XML-Dokumente unterliegen keinen Größenbeschränkungen und stellen daher die verarbeitenden Systeme (z.B. Datenbanksysteme) vor große Herausforderungen.

Für die Suche in relationalen Datenbanksystemen hat sich SQL als Standard-Anfragesprache etabliert und wird von fast allen relationalen Datenbanksystemen unterstützt. Analog hat sich für XML-Dokumente XPath/XQuery [4] als die Standard-Anfragesprache entwickelt. XQuery ermöglicht sowohl die Gewinnung von Information über die Struktur von XML-Dokumenten, als auch die darin gespeicherten textuellen Inhalte (Volltexte). Es ist zu beobachten, dass bei steigender Dokumentengröße die enthaltenen Volltexte in der Regel sehr viel stärker wachsen, als die sie umgebenden strukturellen Informationen. Daher enthalten große Dokumente auch große Volltexte. Um in diesen Volltexten suchen zu können, wurde XQuery durch XQuery Full-Text [2] erweitert.

XQuery Full-Text ermöglicht die Integration komplexer textbasierter Suchbedingungen in XQuery Anfragen und ist Gegenstand aktueller Forschung. Daher existieren bis heute kaum vollständige Implementierungen. Aufgrund der Forschungsrelevanz und der Aktualität von XQuery Full-Text dient es als Grundlage dieser Arbeit. Sie beschreibt die vollständige Integration der XQuery Full-Text Funktionalität in ein bestehendes XML-Datenbanksystem (BaseX [12]).

BaseX wurde am Lehrstuhl für Datenbanken und Informationssysteme der Universität Konstanz entwickelt. Es ist ein natives open source XML-Datenbanksystem, das einen effizienten XQuery Prozessor bietet. Die in dieser Arbeit vorgestellte Integration der XQuery Full-Text Funktionalität wird am Beispiel dieses XQuery Prozessors aufgezeigt. Allerdings sind die erarbeiteten Mechanismen der Indexintegration und Optimierung von Anfragen allgemein gültig und somit auch auf andere Systeme übertragbar.

Die Verarbeitung großer Dokumente bringt mit sich, dass die Inhalte der Dokumente für die Anwender schwer zugänglich sind. Aufgrund der beschränkten Darstellungsfläche ist eine rein textuelle Darstellung des gesamten Inhaltes nicht mehr möglich. Daher sind Visualisierungsverfahren notwendig, die die Inhalte der Dokumente aufbereiten und für den Anwender leicht zugänglich darstellen. Hieran anknüpfend kombiniert und modifiziert diese Arbeit bekannte Visualisierungsverfahren und stellt so einen Ansatz zur Darstellung kompletter Volltexte in ihrem dokumentenbezogenen Kontext vor.

Speziell bei der Suche in Volltexten ist die Hervorhebung der Suchergebnisse in den Volltexten für Anwender von besonderem Interesse. Anwender können so Treffer sehr viel schneller identifizieren und in ihrem Kontext betrachten. Aber auch ein Überblick über die Verteilung der Treffer in den Volltexten wird den Anwendern hierdurch vermittelt. Daher wird im Rahmen dieser Arbeit das interaktive Frontend von BaseX um eine Volltext Visualisierung erweitert, die die gesamten Volltexte darstellt und Suchergebnisse in ihnen visuell hervorhebt.

1.2 Struktur der Arbeit

Zunächst einmal wird ein Überblick über grundlegende Aspekte der Speicherung und Indexierung von XML-Dokumenten in XML-Datenbanksystemen gegeben, dem eine Zusammenfassung der in BaseX implementierten Full-Text Index-Strukturen folgt. Im Anschluss wird ein Verfahren vorgestellt, dass es ermöglicht, Full-Text Index-Strukturen vollkommen automatisch in die Anfrageverarbeitung von Full-Text Anfragen zu integrieren und so erhebliche Performancesteigerungen zu erreichen. Neben der Implementierung und Integration der Full-Text Operatoren, die die XQuery Full-Text Erweiterung beinhaltet, wird detailliert auf die internen Mechanismen eingegangen und eine optimierte Datenhaltung für Full-Text Index-Strukturen vorgestellt. Diesen Betrachtungen schließen sich verschiedene Optimierungsansätze für volltextbezogene Anfragen an.

Abschließend wird ein bestehendes Visualisierungskonzept für XML-Dokumente erweitert, so dass auch die enthaltenen Volltexte vollständig visualisiert und die Treffer von Suchanfragen in den Volltexten farblich hervorgehoben werden. Hierbei findet eine Kombination bekannter Visualisierungsverfahren statt, die jedoch durch Hinzufügen einer Interaktionskomponente zu einem neuen Gesamtkonzept zusammengeführt werden.

KAPITEL 2

Einführung in die XML-Full-Text Funktionalität (XQuery 1.0 und XPath 2.0 Full-Text 1.0)

2.1 Überblick XQuery Full-Text

Im Folgenden wird von einem Suchbegriff gesprochen, dieser kann sowohl aus einem einzelnen Token, als auch aus einer Folge von Tokens (Phrase) bestehen und auch Wildcards enthalten. Ein Token ist als eine Aneinanderreihung von alphanumerischen Zeichen zu sehen.

Die XML Abfragesprache XQuery 1.0 [4] bietet zunächst in ihrer ursprünglichen Form nur sehr begrenzt Unterstützung für inhaltsbezogene Operationen. Es stehen zwar die Funktionen substring(), contains() und matches() zur Verfügung, jedoch berücksichtigen diese die Wichtigkeit eines Knotens hinsichtlich des Resultats einer Anfrage nicht und komplexe textbezogene Anfragen sind hiermit auch nicht möglich. Daher wurde XQuery 1.0 durch die Einführung einer Full-Text Unterstützung erweitert [2]. Diese Erweiterung wird durch die Einführung eines neuen Ausdrucks (*FTContains*) und eine Erweiterung des FLOWR Konstrukts hervorgerufen und bringt die klassische Datenbankwelt und das Information Retrieval zusammen.

Hieraus resultiert die Einführung der *tokenization* als ein neuer Verarbeitungsschritt des Ausführungsmodells, unter der die Unterteilung des gesamten Textes eines Knotens in einzelne kleine Einheiten (*tokens*), die bei einer Full-Text Suchanfrage referenziert werden können, zu verstehen ist. Das Ausführungsmodell (*processing model*) beschreibt den Prozess der Überführung eines XML-Dokumentes in ein XML-Infoset. In diesem Kapitel soll die Full-Text Erweiterung in ihren wesentlichen Bestandteilen motiviert und eingeführt werden. Details zur Full-Text Erweiterung sind der Spezifikation [2] zu entnehmen, die sich allerdings noch in der Entwicklung befindet und möglicherweise zukünftigen Änderungen unterliegt.

2.2 FTContains Expression

Eine FTContains Expression evaluiert eine Sequenz von Knoten gegen eine Full-Text Selektion und gibt einen booleschen Wert zurück [2]. Sie verhält sich analog zu einer

Comparison Expression und ist auch so zu verwenden.

`Syntax:` Expr (`ftcontains` FTSelection FTIgnoreOption?)?

Jeder Knoten aus der Ergebnissequenz von Expr definiert einen Suchkontext und, falls dieser der FTSelection entspricht, wird hierfür *true* zurück gegeben. Die Expr kann sowohl ein einfacher String sein, als auch ein komplexer Pfadausdruck, dessen Ergebnis eine Knotensequenz ist. Die FTSelection kann einen einfachen Suchbegriff, anhand von logischen Operatoren verknüpfte Suchbegriffe (siehe Abschnitt (2.3)) oder auch komplexere positionsbezogene Filterbedingungen (siehe Abschnitt (2.4)) enthalten. Suchbegriffe können zusätzlich durch den Einsatz von Match Options (siehe Abschnitt (2.6)) modifiziert werden. Die optionale FTIgnoreOption ermöglich es, Knoten aus der Ergebnissequenz von Expr zu ignorieren, d.h. auf diesen wird die FTSelection nicht geprüft.

2.3 Logische Full-Text Operatoren

2.3.1 FTOr- und FTAnd-Selection

Die logischen Full-Text Operatoren FTOr und FTAnd verknüpfen zwei FTSelections miteinander und sind analog zu den booleschen Operatorn $\vee$ und $\wedge$ zu sehen. Somit lassen sich z.B. einfache Suchbegriffe, aber auch geschachtelte FTSelections miteinander verbinden, wobei das FTAnd die stärkere Bindung hat (analog zur booleschen Algebra).

2.3.2 FTMildNot

Für den FTMildNot Operator gibt es keinen analogen booleschen Operator, er wird durch die Schlüsselworte "`not in`" beschrieben. Die Selection A `not in` B ist dann erfüllt, wenn eine Sequenz A enthält und A kein Teil von B ist, siehe z.B. **a1**. Falls A und B keinen gemeinsamen Teil besitzen, ist die Anfrage äquivalent zur einfachen Suche nach A.

a1: "a" `ftcontains` "a" `not in` "a a" $\Rightarrow$ *true*

a2: "a a b" `ftcontains` "a" `not in` "a b" $\Rightarrow$ *true*

a3: "a a b" `ftcontains` "a" `not in` "a b" `not in` "a a" $\Rightarrow$ *true*

a4: "a a b" `ftcontains` "a" `not in` "a a" `not in` "a b" $\Rightarrow$ *false*

Anfrage **a2** liefert auch *true*, da das erste "a" in "a a b" die Bedingung erfüllt. Die Auswertung von **a3** kann in zwei Schritte unterteilt werden. Zunächst ist **a2** zu prüfen, dann **a1** und hieraus die Konjunktion zu bilden. Hingegen liefert **a4** *false*, da es in "a a b" kein "a" gibt, das nicht auch in "a a" vorkommt. Der FTMildNot Operator ist eine sinnvolle und interessante Ergänzung der booleschen Standardoperatoren im Zusammenhang mit Textretrieval.

2.3.3 FTUnaryNot

FTUnaryNot ist analog zu dem logischen Operator ¬ zu sehen und wird durch ein "`ftnot`" vor einer FTSelection spezifiziert. Vor jedem FTUnaryNot muss ein weiterer Operator stehen, außer die FTSelection enthält nur einen Suchbegriff. Hieraus folgen beliebige Kombinationen der drei verbleibenden Operatoren mit dem FTUnaryNot. So sind z.B. folgende Anfragen möglich:

a1: "a b c" `ftcontains` "q" `ftor ftnot` "z" ⇒ *true*

a2: "a a b" `ftcontains` "a" `not in` "a b" ⇒ *true*

a3: "a b" `ftcontains` "a" `not in ftnot` "a b" ⇒ *true*

Die Ergebnisse von **a1** und **a2** erscheinen nachvollziehbar und offensichtlich, da "a b c" kein "z" enthält und das erste "a" in "a a b" die Bedingung aus **a2** erfüllt. Bei **a3** wird geprüft ob "a" kein "a b" enthält und im zweiten Schritt ob "a b" ein "a" enthält.

Die W3C Empfehlung lässt eine beschränkte Verwendung des FTUnaryNot Operators zu, so dass dieser immer einem FTAnd folgen muss, bzw. bei FTSelections mit nur einem Suchbegriff darf er alleine stehen. Allerdings wird so die Mächtigkeit von FTUnaryNot stark eingeschränkt.[1] Sie hätte zur Folge, dass die Anfragen **a1** und **a3** nicht mehr der Grammatik entsprechen würden. Im Allgemeinen können Anfragen nicht umformuliert werden, so dass sie trotz der Einschränkungen der Grammatik entsprechen und äquivalent zur ursprünglichen Anfrage sind. Folgendes Bespiel veranschaulicht dies:

a4: ("b", "a z") `ftcontains` "a" `ftand` ("q" `ftor ftnot` "z") ⇒ *false*

a5: ("b ", "a z") `ftcontains` "a" `and` ("b ", "a z") `ftcontains` "q" `ftor ftnot` "z" ⇒ *true*

Ursächlich hierfür ist, dass sich das `ftcontains` auf eine Sequenz von Items bezieht und die `ftcontains` Bedingung für jedes Item der Sequenz geprüft wird. D.h., es existiert kein "a", das nicht in Verbindung mit einem "z" vorkommt, und so liefert die Anfrage **a3** *false*. Für den `and`-Operator hingegen ist es ausreichend, dass bereits ein Item der Sequenz die Bedingung erfüllt. Da es ein Item gibt, das ein "a" enthält, und es ein Item gibt, das ein "q" oder kein "z" enthält, liefert die Anfrage **a5** *true*.

2.4 Positional Filters

Die logischen Full-Text Operatoren ermöglichen es, Suchbegriffe miteinander zu einem Ausdruck zu verbinden. Dieser Ausdruck kann nun um positionsbezogene Bedingungen ergänzt werden, welche in ihrer durch die Anfrage festgelegten Reihenfolge geprüft werden.

FTOrdered Legt durch das Schlüsselwort `ordered` fest, dass die Reihenfolge der Suchbegriffe in der Anfrage und im Ergebniselement gleich sind.

1 In BaseX existiert diese Einschränkung nicht.

FTWindow Ermöglicht die Definition eines Fensters, in dem alle Suchbegriff vorkommen müssen, um in die Ergebnismenge aufgenommen zu werden. Das Fenster kann über eine beliebige Anzahl (N) von FTUnits gespannt werden, wobei eine FTUnit ein Wort[1], Satz[2] oder Absatz[3] sein kann.
`Syntax: window` N FTUnit

FTDistance Distanzen zwischen FTUnits (Worte, Sätze, Absätze) können mittels FTDistance definiert werden, wobei die Distanz sowohl ein fixer Wert(N), als auch ein Bereich(FTRange) sein kann. Über den Bereich lassen sich Maximal- oder Mindestabstände sowie variable Abstände definieren.
`Syntax: distance` FTRange N FTUnit

FTScope Hier kann das Auftreten von Suchbegriffen in gleichen oder unterschiedlichen Sätzen/Absätzen gesteuert werden.
`Syntax:` (`same` | `different`) (`sentence` | `paragraph`)

FTContent Mit FTContent kann festgelegt werden, dass die FTSelection dem Anfang (`at start`) bzw. dem Ende (`at end`) oder dem gesamten Element (`entire content`) entspricht.
`Syntax:` (`at start` | `at end` | `entire content`)

Nun stellt sich die Frage, wie z.B. "A B" `ftor` "C D " `ordered` auszuwerten ist. Hier sagt das W3C klar, dass "A B" `ftor` "C D " `ordered` und "A B" `ordered ftor` "C D " `ordered` äquivalent sind. Die Bindung der FTSelections erscheint somit nicht immer intuitiv.

2.5 Cardinality Selection

Durch FTTimes lässt sich die Häufigkeit der Suchbegriffe in einem Element durch eine in der Anfrage festgelegten Bedingung filtern. Auch hier lässt sich wieder ein Bereich (FTRange) sowie ein fixer Wert (N) als Bedingung definieren.
`Syntax: occurs` FTRange N `times`

2.6 Match Options

Durch den Einsatz von Match Options in einer Anfrage können die Suchbegriffe, die in der Anfrage spezifiziert sind, verändert werden. So kann z.B. ein Suchbegriff aus der Anfrage in Großbuchstaben umgewandelt werden, bevor die FTSelection ausgewertet wird. FTMatchOptions werden nur berücksichtigt, wenn sie explizit in der Anfrage formuliert werden, jedoch sind einige als Standard gesetzt. Sie ermöglichen es, zusätzliche Informationen in Anfragen zu integrieren, die die zugrundeliegenden Dokumente nicht enthalten. Ein Überblick ist der folgenden Tabelle (2.1) zu entnehmen.

1 Folge von alphanumerischen Zeichen
2 Geordnete Menge von tokens und im Sinne der W3C Empfehlung implementationsabhängig; in BaseX begrenzt eines der folgenden Zeichen einen Satz: '.', '!', '?'
3 Geordnete Menge von tokens und im Sinne der W3C Empfehlung implementationsabhängig; in BaseX begrenzt das Zeichen 'Carriage Return' einen Absatz

Tabelle 2.1: Übersicht XQuery Full-Text Match Options

FTMATCHOPTION	*BESCHREIBUNG:*
FTWildCardOption	Verwendung von Wildcards in Suchbegriffen
	'.' genau ein Zeichen anhängen oder einfügen
	'.?' null oder ein Zeichen anhängen oder einfügen
	'.+' mindestens ein Zeichen anhängen oder einfügen
	'.*' beliebig viele Zeichen anhängen oder einfügen
	'.n,m' von n bis m Zeichen anhängen oder einfügen
FTCaseOption	Berücksichtigung von Groß- und Kleinschreibung
	`lowercase` Suchbegriff in Kleinbuchstaben umwandeln
	`uppercase` Suchbegriff in Großbuchstaben umwandeln
	`case sensitive` Groß/Kleinschreibung berücksichtigen
	`case insensitive` Groß/Kleinschreibung nicht berücksichtigen
FTThesaurusOption	Suchbegriffe werden mit passenden Begriffen aus Thesaurus ergänzt
FTStemOption	Begriffe werden auf Basis des Wortstammes verglichen
FTDiacriticsOption	Spezielle Berücksichtigung von diakritischen Zeichen
FTStopWordOption	Eliminierung von Stoppworten aus Anfragen
FTLanguageOption	Festlegung der Sprache für FTStopWords oder FTStemoption
FTExtensionOption	Implementationsabhängige Erweiterung möglich

2.7 Score Variablen und Gewichtungen

Neben dem booleschen Rückgabewert bei dem Vergleich eines Knotens und einer FTSelection können die Full-Text Operatoren, Positional Filter und Match Options auch einen Scoring Wert zurückliefern. Dieser Wert zwischen 0 und 1 beschreibt die Wichtigkeit eines Resultates bezogen auf die zugrundeliegende Suchbedingung. Ein höherer Scoring Wert setzt auch eine höhere Wichtigkeit voraus.

Das Scoring von Resultaten kann durch das Setzen von Gewichtungen (weights) in Full-Text Anfragen beeinflusst werden; diese müssen in der Summe über alle Suchbegriffe den Wert 1 ergeben. Sowohl das verwendete Scoringmodell, als auch die Berücksichtigung solcher Gewichtungen hängt von der jeweiligen Implementation ab.

KAPITEL 3

Grundlegende Konzepte von XML-Datenbanksystemen

3.1 XML-Testdaten

3.1.1 Bibliotheksdaten

Die vorliegende Arbeit verwendet zur Veranschaulichung der vorgestellten Konzepte sowohl Bibliotheksdaten, wie auch verschiedene Instanzen der Wikipedia Enzyklopädie (Auszug siehe Anhang). Die Bibliotheksdaten beinhalten ausschließlich die Metadaten aller Titel der Bibliothek der Universität Konstanz und werden in einem rund 800 MB großen XML Dokument gehalten. Jeder der ca. 1,86 mio Einträge in diesem Dokument fasst alle für den Bibliothekskontext relevanten Daten, wie z.B. Typ des Mediums, Name des Autors, Sprache, Signatur usw. und es sind darin rund 6,5 mio Begriffe enthalten. Die regelmäßige Struktur und die Anbindung von BaseX an MedioVis [15], einem Werkzeug zur visuellen Informationssuche in digitalen Bibliotheken, motivieren die nähere Betrachtung dieses Datensatzes. Die von den MedioVis-Anwendern spezifizierten Suchanfragen werden seit einigen Jahren gespeichert und eine kleine Auswahl hieraus wird in dieser Arbeit verwendet.

3.1.2 Wikipedia Enzyklopädie

Die Wikipedia Enzyklopädie mit ihren umfassenden Full-Texten steht auch als XML-Version zur Verfügung.[1] Ihre ursprüngliche Größe ist für diese Arbeit in Instanzen von 0.1 MB, 1 MB, 10 MB, ..., bis 10 GB unterteilt worden, so dass sich Performanceverläufe bei steigendender Dateigröße analysieren lassen. Neben dem auch über das Internet zugänglichen Full-Text eines jeden Artikels werden in den Dokumenten Titel, Kommentare, Verfasser usw. gespeichert. Die 1 GB Instanz beinhaltet rund 132.000 Artikel, die ca. 23 mio Begriffe enthalten, wobei ein Artikel im Durchschnitt 7.500 Zeichen lang ist.

1 http://de.wikipedia.org/wiki/Wikipedia:Download

3.2 Datenhaltung in XML-Datenbanksystemen

3.2.1 Speicherung von XML-Dokumenten

Nun stellt sich die Frage, wie beispielsweise obige XML-Testdaten effizient in einem XML-Datenbanksystem gespeichert werden können. In der Literatur sind grundsätzlich mindestens zwei Verfahren zu unterscheiden, die im Folgenden beschrieben werden. Die naive Speicherung eines XML-Dokuments in einem Dateisystem unter Ausnutzung der vom Dateisystem gebotenen Funktionalität zur Anfrageverarbeitung erscheint als wenig effizient, da die für XQuery notwendigen Mechanismen i. d. R. nicht vorhanden sind. Der Einsatz spezieller Datenbanksysteme oder die Ergänzung der Funktionalität ist hier unumgänglich. Diese verwenden eine spezielle Adressierung der Knoten von XML-Dokumenten, um deren hierarchische Struktur abzubilden. Die Adressen der Knoten werden zusammen mit weiteren Informationen, wie z.B. über Art und Inhalt des Knotens, gespeichert.

Hierarchische Adressierung - ORDPATH

Eine hierarchische Knotennummerierung wie z.B. ORDPATH [22] hat den Vorteil, dass sie die Struktur des Dokuments in der Knotenadresse schon mit speichert. Sie hat die folgende Struktur:

`rootID.rootChildID.`$\cdots$`.parentID.nodeID`

Die ORDPATH-IDs bestehen aus Integer-Werten, die durch Punkte getrennt sind und haben daher eine variable Länge, abhängig von der Distanz zum Wurzelknoten. Aus diesem Grund werden sie in der Regel komprimiert in einer Index-Struktur gespeichert. Durch die Komprimierung werden die IDs auf eine einheitliche Länge gebracht und können z.B. in logarithmischer Zeit aus B-Bäumen gelesen werden. Allerdings haben Komprimierungsverfahren auch Grenzen und daher lassen sich nicht ohne Weiteres beliebig strukturierte XML-Dokumente speichern.[1] Auch werden Teile der IDs redundant gespeichert, da Kindknoten den gleichen Prefix haben wie ihr Vater. ORDPATH ermöglicht in der Regel effiziente Änderungen an der Baumstruktur. Bei Updates führen globale ORDPATH-IDs möglicherweise zu einem erheblichen Änderungsaufwand aller nachfolgenden Knoten innerhalb des Dokuments.[2] Bei lokalen ORDPATH-IDs kann dieser Änderungsaufwand auf Knoten mit dem gleichen Vater-Knoten verringert werden.[3]

Flache Adressierung

Bei der flachen Adressierung wird zunächst jeder Knoten des Baumes durch eine eindeutige ID (pre), die in einer preorder-Traversierung ermittelt wird, identifiziert. Zusätzlich ist für jeden Knoten eine postorder ID (post) und der pre-Wert des Vaterknoten zu speichern, um

1 Die beliebige Struktur von Dokumenten ist eine der Hauptvorteile von XML im Vergleich zu relationalen Verfahren.
2 Globale ORDPATH-IDs führen zu dokumentenweit eindeutigen IDs.
3 Lokale ORDPATH-IDs sind auf einer Hierarchieebene eindeutig.

alle XPath-Achsenschritte auswerten zu können.[1] Über die Speicherung zusätzlicher Werte, wie z.B. der Distanz zum Vaterknoten, kann die Rekonstruktion des Dokumentenbaumes beschleunigt werden [14].

Allerdings ist das Auffinden von Knoten auf der parent- oder ancestor-Achse mit der pre/post Kodierung nicht so effizient. Da sowohl der pre-, als auch der post-Wert global, d.h. dokumentenweit vergeben werden, ergibt sich bei Updates u.U. ein erheblicher Änderungsaufwand [13]. Daher bietet es sich an, alternativ neben dem pre-Wert (global) auch die relative Distanz zum pre-Wert des Parent-Knotens und die Anzahl der Knoten auf der descendant-Achse (size) zu speichern, die beide lokale Werte sind. Hierbei kann aus der size auch der post-Wert berechnet werden (pre + size = post). Beim Einfügen/Löschen müssen nur die size-Werte aller Knoten auf der ancestor-Achse eines Knotens und die dist-Werte der following siblings der Knoten auf der ancestor-Achse aktualisiert werden. Bei der Speicherung absoluter Werte müssten alle Werte der Datenbanktabelle, die dem eingefügten/gelöschten Knoten folgen, aktualisiert werden.

pre/post/par oder pre/dist/size basierte Knotennummerierung benötigt konstanten Speicherplatz für jede ID unabhängig von der Baumtiefe. Allerdings ist die Baumstruktur nicht mehr direkt aus den Werten abzulesen, sie kann jedoch rekonstruiert werden. Die Suche in einem Dokument kann in einem konstant sequentiellen Zugriff und damit deutlich schneller als bei ORDPATH erfolgen. Hierbei ist es ausreichend, die tabellarische Darstellung des Dokuments sequentielle zu durchlaufen. Allerdings können Änderungen in der Baumstruktur einen erheblichen Aufwand mit sich bringen, der u.U. höher ist als bei ORDPATH. Diese können jedoch durch zusätzliche Datenstrukturen beschleunigt werden. Die Vorteile dieser Adressierung, vor allem bei der effektiven Rekonstruktion der Baumstruktur und dem Speicherverbrauch, überwiegen klar. Nachfolgend ist ein Beispiel zu der pre/dist/size Kontenadressierung aufgeführt.

3.2.2 Datenbanktabelle und Index-Strukturen in XML-Datenbanksystemen

Datenbanktabelle - Basis für Index-Strukturen

Zunächst einmal ist ein Dokument unter Verwendung eines flachen Addressierungsschemas in ein Format zu überführen, das sich in einer Datenbanktabelle speichern lässt, wie bereits im letzten Abschnitt eingeführt. Das nachfolgend links stehende Dokument und die hieraus resultierende Datenbanktabelle Tabelle (3.1) dienen nun als Basis, auf der die vorgestellten Index-Strukturen aufsetzen.

1 Die Parent-Referenz ist für die Achsen child, following-sibling und preceding-sibling notwendig.

```
<w>
   <a>A A A</a>
   <b>B</b>
   <c>C</c>
   <a><b>B B</b><c>C</c></a>
   <b><a>A</a><b>B B</b></b>
   <b>B B</b>
</w>
```

Tabelle 3.1: Datenbanktabelle

PRE	DIST	SIZE	KIND	CONTENT
0	1	20	DOC	doc.xml
1	1	19	ELEM	w
2	1	2	ELEM	a
3	1	1	TEXT	A A A
4	3	2	ELEM	b
5	1	1	TEXT	B
6	5	2	ELEM	c
7	1	1	TEXT	C
8	7	5	ELEM	a
9	1	2	ELEM	b
10	1	1	TEXT	B B
11	3	2	ELEM	c
12	1	1	TEXT	C
13	12	5	ELEM	b
14	1	2	ELEM	a
15	1	1	TEXT	A
16	3	2	ELEM	b
17	1	1	TEXT	B B
18	17	2	ELEM	b
19	1	1	TEXT	B B

Die Datenbanktabelle speichert neben den zur effizienten Rekonstruktion des Dokumentenbaumes notwenigen Spalten `PRE`, `DIST` und `SIZE` weitere Informationen, die hier nur beispielhaft angeführt sind. So wird z.B. die Spalte `KIND` zur Evaluation von nodekind-Tests herangezogen und verhindert so, dass der `CONTENT` von Knoten, die nicht vom Typ `TEXT` sind, bei Full-Text Anfragen durchsucht werden.

Tag Index

Neben der Speicherung des Dokumentenbaumes unter Verwendung eines flachen Adressierungsschemas in einer Datenbanktabelle sind auch die Tags der Elemente, Attribute, genauso wie die Full-Texte von Textknoten zu erfassen. Bei der Suche nach Elementen in der Datenbanktabelle muss diese u. U. komplett durchsucht werden, um alle Knoten zu finden, die einem bestimmten Tag entsprechen. Der Einsatz eines hash-basierten Tag Index beschleunigt diese Suche erheblich, da er zu jedem Tag alle Referenzen der Knoten speichert, die den Tag als Element haben [9]. Zusätzlich ermöglicht er es, weitere Informationen wie z.B. die durchschnittliche Textlänge zu speichern. Ein Tag Index kann auch zur Aufwandsschätzung heran gezogen werden (siehe Kapitel (6.2.2)). Aus Tabelle (3.2) ist der aus dem obigen Dokument resultierende Tag Index zu entnehmen.

Tabelle 3.2: Tag Index

TAG	PRE
b	4, 9, 13, 16, 18
a	2, 8, 14
c	6, 11
w	1

Attribut Index

Darüber hinaus können Elemente durch die Verwendung von Attributen mit Zusatzinformationen angereichert werden. Jedes Attribut besteht immer aus einem Attributname/-wert Paar. Da Attribute eigenständige Knoten im Dokumentenbaum sind, werden sie nicht im Tag Index gespeichert. Sie können nur über die Attribut-Achse und nicht über die Child-Achse angesprochen werden. Häufig sind Attribute nummerische Werte, wie z.B. IDs oder Jahreszahlen. Daher bietet es sich an, diese in einer baumbasierten Index-Struktur zu speichern, die auch eine effiziente Verarbeitung von Bereichsanfrage ermöglicht [9].

Name Index

Die Trennung von Tag und Attribut Index ist nicht zwangsläufig zweckmäßig und zwingend notwendig. Vielmehr stellt sich die Frage, in wieweit es sinnvoll ist, komplette Inhalte von Elementen in ihrer ursprünglichen Form in einer Index-Struktur zu ihrem umschließenden Tag, der hierfür als Schlüsselwert dient, zu speichern. Hierdurch kann bei den Suchanfragen, die sich auf den gesamten Inhalt beziehen, sicherlich eine Effizienzsteigerung erzielt werden. Bei Anfragen, die nur Teile des Inhaltes spezifizieren, ist die Index-Stuktur nicht verwendbar. Darüber hinaus benötigen solche Strukturen sehr viel Speicher, da sie im grunde genommen das gesamte Dokument beinhalten.

Aus diesen Gründen erscheint es sinnvoll, nur die in dem Dokument vorkommenden Tags in einer hashbasierten Struktur zu speichern, so dass deren Hashwerte anstelle der ursprünglichen Tags in der Datenbanktabelle gespeichert werden können. Dies hat den Vorteil, dass die Hashwerte im Vergleich zu den unterschiedlich langen Tags konstanten Speicher benötigen. Allerdings ergibt sich auch ein geringer zusätzlicher Aufwand, da die Berechnung von Hashwerten sehr effizient erfolgt. Des Weiteren können statistische Daten, die die Anfrageverarbeitung beschleunigen können, gespeichert werden.[1] Attribute können nach dem gleichen Prinzip gespeichert werden. Der in Tabelle (3.3) gezeigte Name Index korrespondiert mit dem obigen Dokument und der Datenbanktabelle (Tabelle (3.1)). Jedoch werden hier die Tags in ihrer Ursprungsform und nicht als Hashwerte dargestellt, um die Lesbarkeit der Tabellen zu erhöhen - real werden, wie bereits erwähnt, in der Spalte `CONTENT` der Datenbanktabelle (Tabelle (3.1)) und in der Spalte `VAL` der Name Index Tabelle (3.3) Hashwerte gespeichert.

1 z.B. die Häufigkeit oder die Anzahl einzigartiger Werte

Tabelle 3.3: Name Index

NAME	FREQUENCY	NUM. VALUES	AVG. CHARS
b	5	2	2.4
a	3	2	3.66
c	2	1	1.0
w	1	0	0

Neben dem Tag (`NAME`) speichert der Name Index auch noch die Häufigkeit (`FREQUENCY`), wie oft der Tag in dem Dokument vorkommt, die Anzahl der einzigartigen Werte (`NUM. VALUES`) und deren durchschnittliche Länge (`AVG. CHARS`). Diese Felder werden zur Beschleunigung der Anfrageverarbeitung verwendet.

Value Index

Da im Name Index keine Inhalte von Knoten gespeichert werden, können diese in einem getrennten Value Index gesammelt verwaltet werden. Der Einsatz baumbasierter Strukturen erscheint hier sinnvoll, da sowohl Bereichsanfragen wie auch die exakte Suche möglich sein sollte. Gespeichert wird hier der gesamte Inhalt eines Knotens, wie z.B. Attributwerte, aber auch ganze Texte. Da nur eine exakte Suche auf dem gesamten Inhalt des Knotens bzw. Bereichsanfragen möglich sind, ist es sinnvoll, eine Größenbeschränkung für die Inhalte von Elementknoten einzuführen. Eine komplette Speicherung sehr langer Inhalte ist somit nicht zweckmäßig. In diesen Fällen ist ein spezieller Full-Text Index anzulegen, der diese Inhalte fasst. In Tabelle (3.4) ist der Value Index zu dem oben angeführten Dokument zu sehen, wobei `VALUE` den Inhalt des Knoten fasst und dessen Häufigkeit der Spalte `FREQUENCY` zu entnehmen ist.

Tabelle 3.4: Value Index

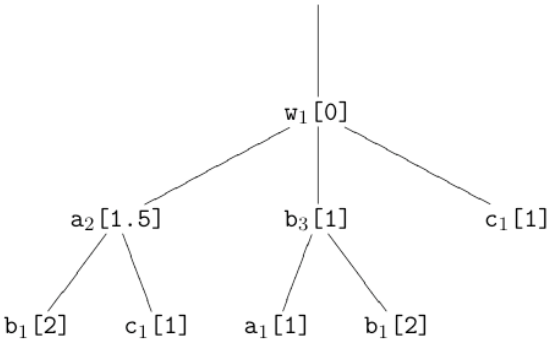
VALUE	FREQUENCY
B B	3
C	2
B	2
A A A	1
A	1

Path Summary

Um Informationen über die Struktur eines Dokuments zusammengefasst und schnell zugänglich zur Verfügung zu stellen, ist der Einsatz einer Path Summary zweckmäßig. Sie fasst jeden einzigartigen Pfad von der Wurzel bis zu einem Blattknoten in einer baumbasierten Struktur zusammen. Je regulärer die Struktur des Dokumentes ist, desto kompakter fällt die Path Summary hierfür aus. Im Abschnitt (6.2.2) wird die Path Summary zur Aufwandsabschätzung herangezogen - sie kann aber auch an anderer Stelle zur Optimierung der Anfrageverarbeitung verwendet werden. Nachfolgend ist in Abbildung (3.1) die Path

Summary für das obige Dokument abgebildet.

Abbildung 3.1: Path Summary: t_n[k], t = Tag, n = Häufigkeit, k = ϕTextlänge



Full-Text Index

Wie bereits ausgeführt, enthalten Full-Text Index-Strukturen die gesamten Full-Texte eines Dokuments. Allerdings speichern sie nicht den gesamten Text je Knoten, sondern die Begriffe der Knotentexte. Hierbei kann eine Stoppwortreduktion genutzt werden, um die Index-Stuktur Größe zu reduzieren. Aufgrund der umfangreichen Full-Text Funktionalität von XQuery, die exakte Suche, Wildcards, Stemming, usw. ermöglicht, ist es durchaus zweckmäßig, mehrere speziell auf einzelne Suchverfahren hin optimierte Index-Strukturen zu verwenden. Im Folgenden wird speziell auf die Full-Text Index-Strukturen genauer eingegangen.

3.3 Full-Text Index-Strukturen

3.3.1 Adressierung von Inhalten eines Knotens

Wie bereits eingeführt, speichern der Tag Index und der Attribut Index sowohl strukturbasierte, wie auch inhaltsbezogene Informationen eines XML-Dokumentes, während eine Full-Text Index-Struktur den Inhalt der Textknoten umfasst. Die Knotenreferenz wird als Information zu jedem Token des zuvor in Tokens zerlegten Knotentextes (siehe Abschnitt (2.1)) indexiert, so dass die Tokens als Suchschüssel fungieren. Damit können bei der späteren Suche nach einem Token die entsprechenden Knoten, in denen das Token enthalten ist, wieder referenziert werden.

Für die Verarbeitung von Full-Text bezogenen Anfragen, die nicht nur logische Full-Text Operatoren (siehe Abschnitt (2.3)), sondern auch Positional Filters (siehe Abschnitt (2.4)) verwenden, reicht es nicht aus, nur zu wissen, welches Token in welchen Knoten vorkommt. Vielmehr sind Informationen über die Position der Tokens in dem Knotentext notwendig. So ist z.B. für ein FTOrdered die Tokenposition notwendig, um das Ordnungskriterium prüfen zu können.

Zunächst können zwei Arten der Positionsermittlung unterschieden werden. Zum einen die Verwendung von globalen, d.h. über das gesamte Dokument hin gültige Positionswerte oder lokale, deren Gültigkeit auf den Knoten beschränkt ist. Bei globalen Positionswerten wird der gesamte Full-Text des Dokuments, Token für Token, knotenübergreifend fortlaufend nummeriert. Eine solche Nummerierung wird vor allem in Verbindung mit einer ORDPATH Adressierung (siehe Abschnitt (3.2.1)) der Knoten eingesetzt [16]. Der Positionswert wird als weitere Zahl direkt an die Knotenadresse, getrennt durch einen ”.”, angehängt. Dieses Verfahren hat vor allem bei der Atomisierung von Knoten[1] den Vorteil, dass die Positionswerte ihre Gültigkeit behalten. Allerdings entstehen bei Dokumenten mit langen Volltexten sehr lange Positionswerte, die viel Speicher benötigen. Hier kann durch Komprimierung versucht werden, den Speicherbedarf jedes Positionswertes zu senken, wobei die Komprimierung bei sehr großen Werten kaum noch Einsparungen hervorruft. Hinzu kommt, dass die Komprimierung sowie die Dekomprimierung zusätzlichen Overhead generiert. Anfragen, bei denen explizit der Anfang eines Knotentextes durchsucht werden soll (FTScope Bedingung siehe Abschnitt (2.4)), können nicht ohne Zusatzinformationen ausgewertet werde, da aus dem absoluten Positionswert die tatsächliche Position des Tokens in dem Knotentext nicht hervorgeht. Das Hauptproblem globaler Positionswerte entsteht durch Updates, bei denen der Full-Text von Knoten betroffen ist, da hier alle nachfolgenden Positionswerte ungültig werden und deren Aktualisierung einen erheblichen Änderungsaufwand verursacht.

Lokale Positionswerte, bei denen die Token des Knotentextes für jeden Knoten von null beginnend nummeriert werden, bringen den Vorteil mit sich, dass selbst bei sehr großen Dokumenten kleine und damit kurze Positionswerte entstehen. Ihr Speicherverbrauch lässt sich durch Komprimierung weiter reduzieren. Es lassen sich so ohne weitere Informationen alle Full-Text Anfragen evaluieren. Allerdings verlieren die Werte bei der Atomisierung ihre Gültigkeit, können jedoch auf einfache Weise angepasst werden. Bei Full-Text bezogenen Updates müssen nur die Positionswerte der nachfolgenden Tokens des Knoten Full-Textes geändert werden. Der Änderungsaufwand ist erheblich geringer als bei der globalen Adressierung.

Allerdings hat der Einsatz von Full-Text Index-Strukturen und damit auch die Adressierung von Inhalten der Knoten Grenzen. Aufgrund von Atomisierung können neue Worte entstehen, so wird z.B. aus "<w>b<x>b</x></w>" nach der Atomisierung "bb". Eingesetzte Full-Text Indizes müssten diese Begriff beim Parsen des Dokumentes erkennen und zusätzlich speichern, um eine Suchanfrage nach "bb" überhaupt korrekt beantworten zu können, unabhängig von der Art des Positionswertes. Die Fragestellungen, die sich durch die Atomisierung bzw. Updates ergeben, werden in dieser Arbeit nicht weiter betrachtet. Abschließend lässt sich festhalten, dass der Speicherverbrauchsvorteil aufgrund der Vergabe lokaler Positionswerte diesem Konzept den Vorzug gibt. Im Folgenden werden die in BaseX eingesetzten Index-Strukturen vorgestellt und miteinander vergleichen.

1 D.h. aus dem Full-Text eines Knotens werden alle Tags der Knoten auf der descendant Achse entfernt und ein Full-Text daraus gebildet. Bsp.: aus "<w>a b <x>c</x> d" wird durch Atomisierung "a b c d"

3.3.2 Eingesetzte Index-Strukturen in BaseX

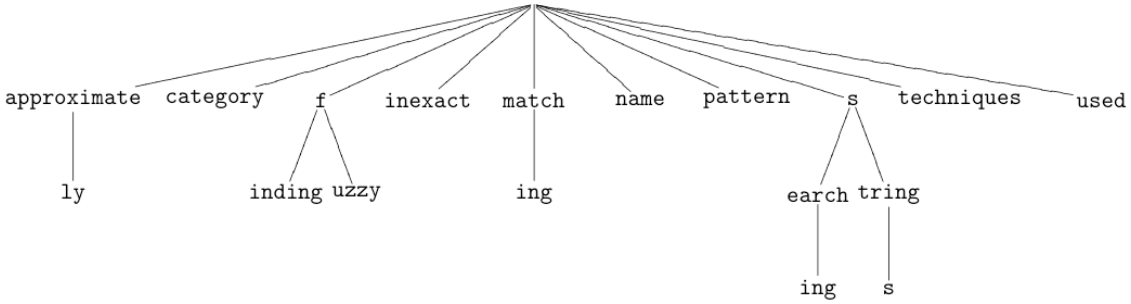
An dieser Stelle werden die zwei in BaseX integrierten Full-Text Index-Strukturen vorgestellt, die beide zwar für spezielle Anwendungsfälle optimiert sind, aber auch eine sehr effiziente exakte Suche ermöglichen. Zum einen handelt es sich um einen Compressed Trie [8], der als Standard Full-Text Index dient, zum anderen um einen speziellen Fuzzy Index, der entwickelt wurde, um effiziente unscharfe Suche in Full-Texten zu ermöglichen. Beide Index-Stukturen sind im Rahmen dieser Arbeit entwickelt und implementiert worden.

Der Compressed Trie ist modifiziert aus der ursprünglich hauptspeicherbasierten Optimierung auf Festplatte übertragen worden und wird in einer Tabellenstruktur gespeichert [3, 20]. Diese Struktur erlaubt eine effiziente exakte Suche, Bereichsanfragen, Wildcards und auch eine unscharfe Suche. Der Fuzzy Index, motiviert durch die Anbindung von BaseX an MedioVis [15], basiert auf einer zweistufigen Sortierung aller Tokens des Full-Textes, zunächst nach Tokenlänge, dann alphanumerisch.

```
<page>
   <title>Fuzzy string searching</title>
   <text>Fuzzy string search is the name that is used for a category of techniques
for string searching/finding strings that approximately match some given pattern
string. It may also be known as approximate or inexact matching.
   </text>
</page>
```

Das obige Dokument-Fragment wird nach der Stopwortbereinigung in die Index-Strukturen eingefügt, der daraus resultierende Aufbau im Folgenden veranschaulicht und die Funktionsweise der Index-Strukturen erläutert.

Abbildung 3.2: Compressed Full-Text Trie ohne Stoppworte - als Baum dargestellt



Die Speicherung des Compressed Trie in der in Abbildung (3.2) gezeigten Form bestehend aus Knotenobjekten, die durch Kanten miteinander in Verbindung stehen, benötigt sehr viel Speicher, da Objekte und nicht atomare Datentypen zu speichern sind. Daher empfiehlt es sich, die Baumstruktur in eine Tabelle zu überführen, wobei die Kante eines Knotens durch das Speichern der ID aller Kindknoten bei dem Vaterelement nachgebildet wird. Zu jeder ID wird zusätzlich noch das erste Zeichen des Kind-Tokens gespeichert, so

dass die Traversierung des Compressed Trie beschleunigt wird. Die Tabelle (3.5) veranschaulicht den Aufbau und zeigt zusätzlich noch die gespeicherten Full-Text Daten (pre-, pos-Werte).

Tabelle 3.5: Compressed Full-Text Trie ohne Stoppworte - in Tabellenform

ID	*TOKEN*	*KINDER*	*PRE*	*POS*
0	1,a, 3,c, 4,f,	7,i, 8,m,	10,n, 11,p,	12,s, 17,t, 18,u
1	approximate	2,l	5	15
2	ly		5	11
3	category		5	5
4	f	6,i, 5,u		
5	uzzy		3,5	0,0
6	inding		5	9
7	inexact		5	16
8	match	9,i	5	12
9	ing		5	17
10	name		5	3
11	pattern		5	13
12	s	15,e, 14,t		
13	ing		3,5	2,8
14	tring	16,s	3,5,5,5	1,1,7,14
15	earch	13,i	5	2
16	s		5	10
17	techniques		5	6
18	used		5	4

Bei der Suche nach "match" muss nun nur noch der Wurzelknoten von Festplatte gelesen, dann binär nach "m" durchsucht und abschließend das Kind mit der ID 8 gelesen werden. Bei einem Standard Compressed Trie müssten alle Kinder des Wurzelknotens von Festplatte gelesen werden, bis das Kind mit der ID 8 gefunden ist. Mit dem pre-Wert wird der Textknoten, in dem das Token vorkommt, referenziert. Der pos-Wert gibt die relative Position des Tokens im Text des Knotens an. Neben dem Verweis auf alle Kindknoten eines Blattes wird in der festplattenorientierte Variante auch noch der jeweilige Anfangsbuchstaben des dort gespeicherten Tokens im Vater mit abgelegt. Damit kann, wie oben beschrieben, die Suche nach Tokens effizienter durchgeführt werden.

Tabelle 3.6: Fuzzy Index - Tokenlänge und ID

TOKENLÄNGE	ID
4	0
5	2
6	4
7	6
8	10
9	12
10	13
11	14
13	15

Tabelle 3.7: Fuzzy Index - sortiert nach Tokenlänge und alphanumerisch

ID	TOKEN	PRE	POS
0	name	5	3
1	used	5	4
2	fuzzy	3,5	0,0
3	match	5	12
4	search	5	2
5	string	3,5,5,5	1,1,7,14
6	finding	5	9
7	inexact	5	16
8	pattern	5	13
9	strings	5	10
10	category	5	5
11	matching	5	17
12	searching	3,5	2,8
13	techniques	5	6
14	approximate	5	15
15	approximately	5	11

Der Fuzzy Index besteht aus einer Tabelle (3.7), die zunächst nach der Tokenlänge, dann alphanumerisch sortiert ist. Eine weitere Tabelle (3.6) verweist für jede Tokenlänge auf das erste Token mit dieser Länge. Die Sortierung nach der Tokenlänge beschleunigt die unscharfe, wie auch die exakte Suche. Bei der exakten Suche, kann über die Wortlänge das erste und letzte Token dieser Länge ermittelt und in diesem Bereich binär gesucht werden. Bei einer unscharfen Suche, z.B. nach "match" mit einem zugelassenen Fehler, müssen nur Tokens der Länge 4 bis einschließlich der Länge 6, d.h. die Einträge ab der ID = 0 bis einschließlich der ID = 5 in Betracht gezogen werden.

Bei der unscharfen Suche auf dem Compressed Trie (Tabelle (3.5)) müssten zunächst alle Kinder des Wurzelknotens überprüft werden und bei den Knoten mit der ID = 4 und der ID = 12 auch noch deren Kinder. Zusätzlich müsste noch das Kind des "match" Knotens geprüft werden.

Tabelle 3.8: Laufzeit und Hauptspeicherverbrauch zum Generieren der Index-Strukturen

MESSWERT	INDEX	1 MB	10 MB	100 MB	1 GB	10 GB
Zeit (sek)	F	0,5	2	16	166	1.680,9
	CT	0,8	2,7	20,7	184,2	2.857,8
Speicher (mb)	F	4,4	23,6	161	1.207	10.886
	CT	6,9	33	207	1.525	13.598

Die Tabelle (3.8) zeigt den Hauptspeicherverbrauch für die Generierung des Fuzzy

Index (F) und des Compressed Trie (CT) für unterschiedliche Wikipedia Instanzen. Es zeigt sich sehr deutlich, dass der Fuzzy Index schneller generiert wird und auch weniger Hauptspeicher benötigt. Das Zusammenfassen von gleichen Präfixen vermeidet die Mehrfachspeicherung, allerdings muss die Baumstruktur des Compressed Tries mit gespeichert werden. Dies benötigt mehr Speicher, als die Komprimierung einspart. Die Speicherung der Full-Text Daten benötigt in beiden Fällen genau gleich viel Platz, da sie in der gleichen Form geschieht.

Abbildung 3.3: Performancevergleich exakte Suche Fuzzy Index und Compressed Trie

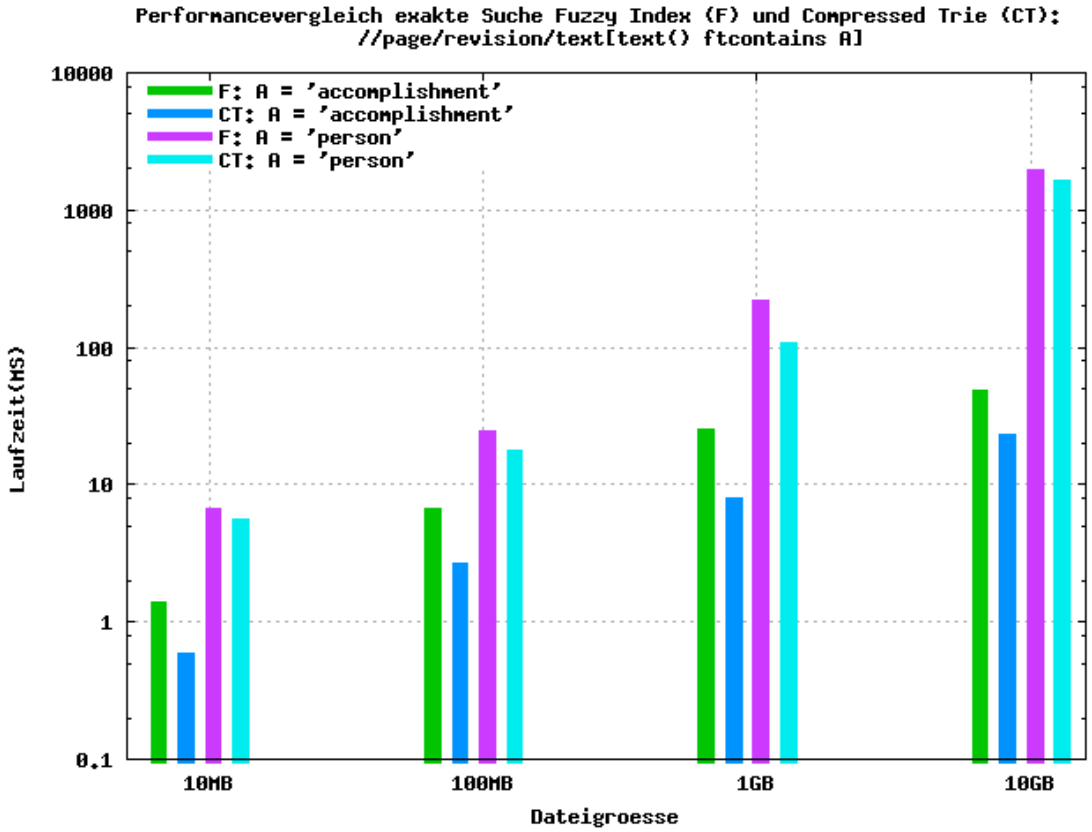


Tabelle 3.9: Anzahl Treffer exakte Suche

	10 MB	*100 MB*	*1 GB*	*10 GB*
accomplishment	4	31	239	2470
person	119	1174	9891	112417

Bei der Performance der Anfrageverarbeitung zeigt sich in Abbildung (3.3), dass der Compressed Trie leichte Vorteile gegenüber dem Fuzzy Index bei der exakten Suche hat. Die Anfragezeiten liegen alle im interaktiven Bereich und zeigen die Effizienz der Strukturen auch bei einer größeren Anzahl von Treffern (Tabelle (3.9)).

Abbildung 3.4: Performancevergleich exakte Suche Fuzzy Index und Compressed Trie

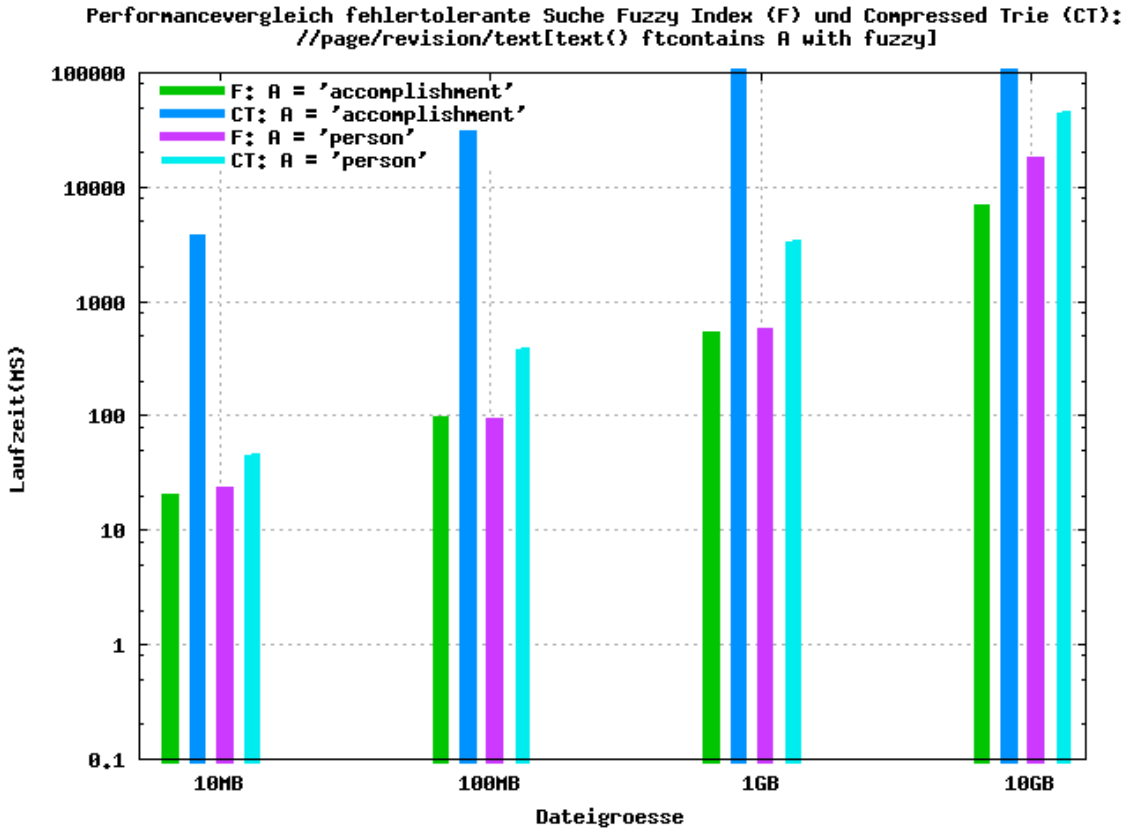


Tabelle 3.10: Anzahl Treffer unscharfe Suche

	10 MB	*100 MB*	*1 GB*	*10 GB*
accomplishment	31	290	2413	23955
person	141	1475	12854	147880

Der Fuzzy Index gewährleistet darüber hinaus auch noch eine effiziente unscharfe Suche. Wie erwartet, zeigt sich in Abbildung (3.4), dass diese langsamer als die exakte Suche aber effizienter, als die unscharfe Suche auf dem Compressed Trie ist. Bei steigender Anzahl an zugelassenen Fehlern und zunehmender Suchbegriffslänge führt die Baumstruktur des Compressed Trie zu einem erheblichen Traversierungsaufwand, der ursächlich für die schlechtere Performance ist. Für die 1 GB und 10 GB Instanzen liegt seine Performance über 100.000 ms und die entstehenden Treffermengen sind Tabelle (3.10) zu entnehmen.

Sollten sich auf grund von Updates inhaltliche bzw. strukturelle Änderungen an indexierten Dokumenten ergeben, so verlieren die vorgestellten Strukturen aktuell noch ihre Gültigkeit. Zukünftigen Arbeiten werden Updates auf den Strukturen ermöglichen.

KAPITEL 4

Verarbeitungsmodi für das Evaluieren von Pfadausdrücken mit Full-Text Erweiterung

4.1 Auswertung von Pfadausdrücken mit Full-Text Erweiterung

In diesem Kapitel wird zunächst grundlegend auf die Auswertung von XQuery Anfragen eingegangen. Im Folgenden werden die speziellen Besonderheiten bei der Auswertung von XQuery Full-Text Anfragen betrachtet. Der Auswertungprozess kann in die Schritte Parsing, Kompilierung und Evaluation unterteilt werden. Im Parsingschritt wird die gestellte Anfrage sequentiell gelesen und gegen eine die Anfragesprache definierende Grammatik (*XQuery Query Grammar*) geprüft. Der Parser implementiert diese Grammatik und baut aus der Anfrage und den Regeln der Grammatik einen Expression Tree auf, der die Anfrage repräsentiert [1]. Die Knoten in diesem Baum entsprechen den Expressions der Anfrage, die Kanten repräsentieren die Datenströme. Ist die Anfrage kein gültiger Ausdruck im Sinne der Grammatik, entsteht ein Fehler und die Anfrage kann nicht verarbeitet werden. Hier kann der Parser dem Anwender eine Rückmeldung über mögliche Lösungsvorschläge geben, die allerdings in Anbetracht der Komplexität der Grammatik nicht immer zu eindeutigen Vorschlägen führen muss.

Ausgehend vom Expression Tree werden im Kompilierungschritt z.B. Typchecks und Optimierungen durchgeführt, die den Expression Tree in einen Iterator Tree überführen. Die Implementierung der Iteratoren bildet die Funktionalität, die mit den Expressions in Verbindung steht, ab und evaluiert diese im nächsten Ausführungsschritt.

In diesem Kapitel wird auf den Kompilierungsprozess und die hier möglichen Optimierungen für Pfadausdrücke mit Full-Text Erweiterung detailliert eingegangen und gezeigt, wie unter Ausnutzung einer Index-Struktur diese Klasse von Anfragen optimiert und so effizient verarbeitet werden kann.

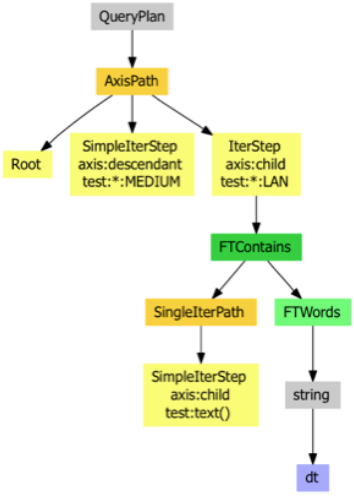
4.2 Sequentieller Verarbeitungmodus

Wird nach dem in Abschnitt (4.1) gezeigten dreistufigen Verfahren vorgegangen, ergibt sich für die Anfrage `//MEDIUM/LAN[text() ftcontains "dt"]` der in Abbildung (4.1) gezeigte Expression Tree. Er zeigt gelb eingefärbt die pfadbezogenen Operatoren und grün

eingefärbt die inhaltsbezogenen Operatoren.

Abbildung 4.1: Auszug XQuery Full-Text Grammatik und Expression Tree von Anfrage `//MEDIUM/LAN[text() ftcontains "dt"]`

```
...
FTContainsExpr          ::=     RangeExpr ( "ftcontains" FTSelection FTIgnoreOption? )?
...
FTSelection             ::=     FTOr FTPosFilter* ("weight" RangeExpr)?
FTOr                    ::=     FTAnd ( "ftor" FTAnd )*
FTAnd                   ::=     FTMildNot ( "ftand" FTMildNot )*
FTMildNot               ::=     FTUnaryNot ( "not" "in" FTUnaryNot )*
FTUnaryNot              ::=     ("ftnot")? FTPrimaryWithOptions
FTPrimaryWithOptions    ::=     FTPrimary FTMatchOptions?
FTPrimary               ::=     (FTWords FTTimes?) | ("(" FTSelection ")")
                                | FTExtensionSelection
FTWords                 ::=     FTWordsValue FTAnyallOption?
FTWordsValue            ::=     Literal | ("{" Expr "}")
...
FTMatchOptions          ::=     FTMatchOption+
FTMatchOption           ::=     FTLanguageOption | FTWildCardOption
                                | FTThesaurusOption | FTStemOption
                                | FTCaseOption | FTDiacriticsOption
                                | FTStopWordOption | FTExtensionOption
...
```



Aus den Produktionsregeln der Grammatik (Abbildung (4.1)) ist sehr gut zu erkennen, dass jeder FTContainsExpr eine FTSelection folgen muss, jedoch die FTIgnoreOption optional ist. Die Grammatik leitet zu einem FTPrimary weiter, dem ein FTWords und eine optionale FTMatchOption folgen muss. Das FTWords ist in dem Expression Tree zu erkennen. Es hat ein Literal vom Typ String als Argument und dieses wiederum den Werte "dt".

Die eigentliche Ergebnisberechnung der Anfrage vollzieht sich nun in mehreren Schritten. Das *: vor den Elementen LAN und MEDIUM ist der namespace prefix, wobei *:/* der Standard namespace ist.[1]

1. Anfrage mit Tag: `*:MEDIUM`
2. Achsentest: `descendant`; Elementtest: `*:MEDIUM`
   Für jeden Knoten aus 1. sind alle Knoten der descendant-Achse zu ermitteln.
3. Achsentest: `child`; Elementtest: `*:LAN`
   Für jeden Knoten aus 2. sind alle Knoten der child-Achse zu ermitteln, die den Tag LAN besitzen.
4. Achsentest: `child`; Typtest: `text()`
   Für jeden Knoten aus 3., der auf der child-Achse liegt und ein Textknoten ist, ist sein Text aus der Datenbank zu laden.

1 In dieser Arbeit werden keine namespaces verwendet, daher erscheint immer der Standard namespace vor jedem Tag. Mittels namespaces kann die Sichtbarkeit von Elementen oder Funktionen beschränkt werden.

5. Jeder Text aus 4. wird sequentiell nach dem String "a" durchsucht und, falls vorhanden, in die Ergebnismenge aufgenommen.

Die Laufzeit der Anfrageauswertung hängt somit hauptsächlich von der Häufigkeit des Tags Medium (Schritt 1), wie auch von der Textlänge (Schritt 5) der zu ladenden Textkonten (Schritt 4) ab. In diesem Beispiel handelt es sich um Bibliotheksdaten, die eine sehr regelmäßige Struktur haben und zu jedem Medium ein Sprachelement (LAN) hinterlegt ist. Damit schränkt der Indexzugriff die Treffermenge kaum ein (geringe Selektivität), ebenso wie der Elementtest (LAN) im dritten Schritt. Die Dokumentengröße hat hier einen entscheiden Einfluss auf die Performance dieser Anfrage.

Im Folgenden wird gezeigt, wie sich diese Klasse von Anfragen unter Einbeziehung weiterer Index-Strukturen optimieren lässt und sich so die Laufzeit maßgeblich reduziert.

4.3 Indexbasierter Verarbeitungsmodus

4.3.1 Anbindung der Index-Struktur zur Anfrageverarbeitung

Anbindung mit Pfadumkehrung

In relationalen Datenbanksystemen hat sich der Einsatz von Index-Strukturen zur effizienten Anfrageverarbeitung bewährt [10]. Hier entscheidet das System anhand einer Kostenabschätzung, ob ein Indexzugriff zweckmäßig ist, und passt den Anfrageplan entsprechend an. Ziel ist es, zunächst einen möglichst selektiven Indexzugriff, der wenige Zwischenresultate erzeugt, zu haben und hierauf aufbauend weitere Berechnung folgen zu lassen. Dieses Vorgehen soll nun auf XML-Full-Text Anfragen übertragen werden.

Im Folgenden werden zunächst nur Anfragen betrachtet, die ein Prädikat enthalten, in denen ein FTContains Ausdruck mit einem voranstehenden Pfadausdruck vorkommt, wie z.B. die Anfrage `//MEDIUM/LAN[text() ftcontains "dt"]`. Anstelle des Pfadausdrucks ist aufgrund der Grammatik auch eine beliebige Expression möglich.[1] Diese Filter Expression bleibt zunächst unberücksichtigt, da die Pfadumkehrung hier nicht angewendet werden kann. In Abschnitt (4.2) ist bereits die sequentielle Verarbeitung der obigen Anfrage ausführlich beschrieben worden. Es wird zunächst der Pfadausdruck `//MEDIUM/LAN` und dann das Prädikat `[text() ftcontains "dt"]` mit dem Full-Text Teil der Anfrage ausgewertet. In der Arbeitsgruppe ist ein Verfahren entwickelt worden, bei dem Full-Text Anfragen zunächst auf den Full-Text Index zugreifen und dann der Pfadausdruck ausgewertet wird. Allerdings muss dieser dann invertiert werden, da nicht mehr von der Wurzel zum Blattknoten durch den Dokumentenbaum traversiert wird, sondern vom Blattknoten zurück zur Wurzel. Die Achsen sind nach Tabelle (4.1) zu invertieren.

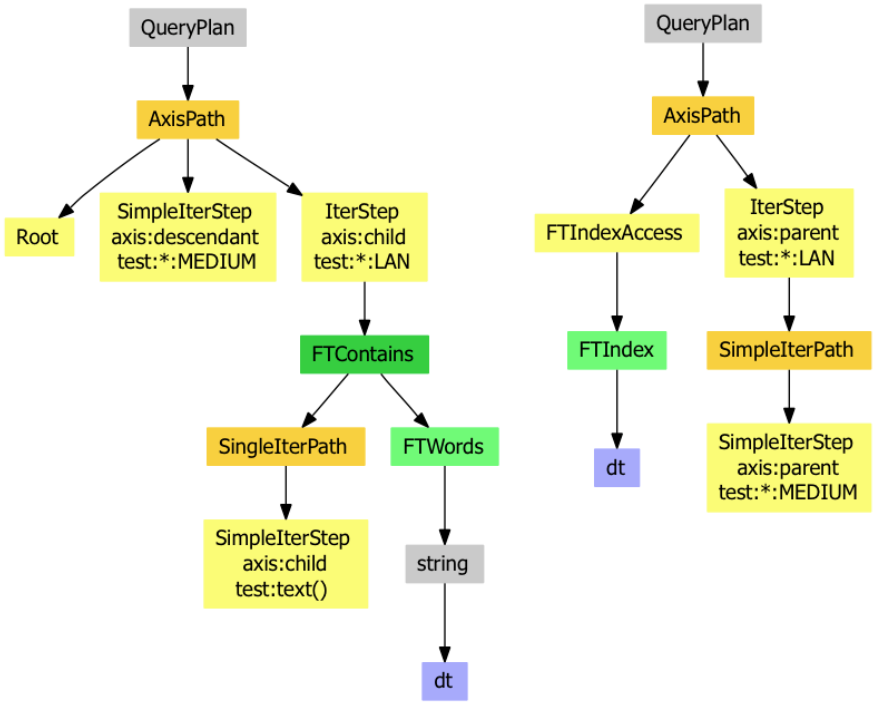
1 z.B. `let $a := "a b c" return $a ftcontains "a"`

Tabelle 4.1: Übersicht von Achsen und entsprechend invertierten Achsen

ACHSE	INVERTIERTE ACHSE
ancestor	descendant
ancestor-or-self	descendant-or-self
attribute	parent
child	parent
following-sibbling	preceding-sibbling
following	preceding
self	self

Zunächst ist der Full-Text Teil `LAN[text() ftcontains "dt"]` der Anfrage zu invertieren. Der Index liefert für das "dt" die Referenz auf Textknoten, nicht auf das parent Element `LAN`. Daher muss ein `parent::*:LAN` nach dem Indexzugriff ausgewertet werden, um von dem Textknoten auf den Elementknoten zu navigieren. Da die Anfrage `//MEDIUM/LAN[text() ftcontains "dt"]` als Ergebnismenge `LAN`-Knoten hat, darf der Kontext nun nicht mehr verändert werden. Daher sind alle weiteren Schritte in einem Prädikat zu sammeln. Nun folgt die Invertierung aller dem Indexzugriff voranstehenden Schritte. Aus `MEDIUM/LAN` (/LAN ≡ child::LAN) wird `[parent::*:Medium]` entsprechend der Tabelle (4.1). Die interne Repräsentation der Anfrage ist Abbildung (4.2) zu entnehmen.

Abbildung 4.2: Anfrageplan ohne Index (links) und mit Indexzugriff (rechts) für die Anfrage `//MEDIUM/LAN[text() ftcontains "dt"]`



Die Anfragepläne in Abbildung (4.2) zeigen sehr deutlich die Auswirkung der Pfadum-

kehrung auf die Verarbeitung der Anfrage. Ohne Indexzugriff wird zunächst der Pfad von dem Wurzelknoten aus bis in den Textknoten als Kind eines LAN-Knotens verfolgt und dort nach "dt" gesucht. Im rechten Anfrageplan liefert der Indexzugriff eine Menge von Textknoten, die um die Knoten, die dem in der Anfrage festgelegten Pfad nicht entsprechen, bereinigt werden. Abbildung (4.3) stellt die Performance der beiden Varianten beispielhaft dar. Hierbei handelt es sich um Instanzen der Wikipedia.xml in der Größe von 1 MB bis zu 1 GB.

Abbildung 4.3: Performancevergleich sequentielle und indexbasierte Verarbeitung

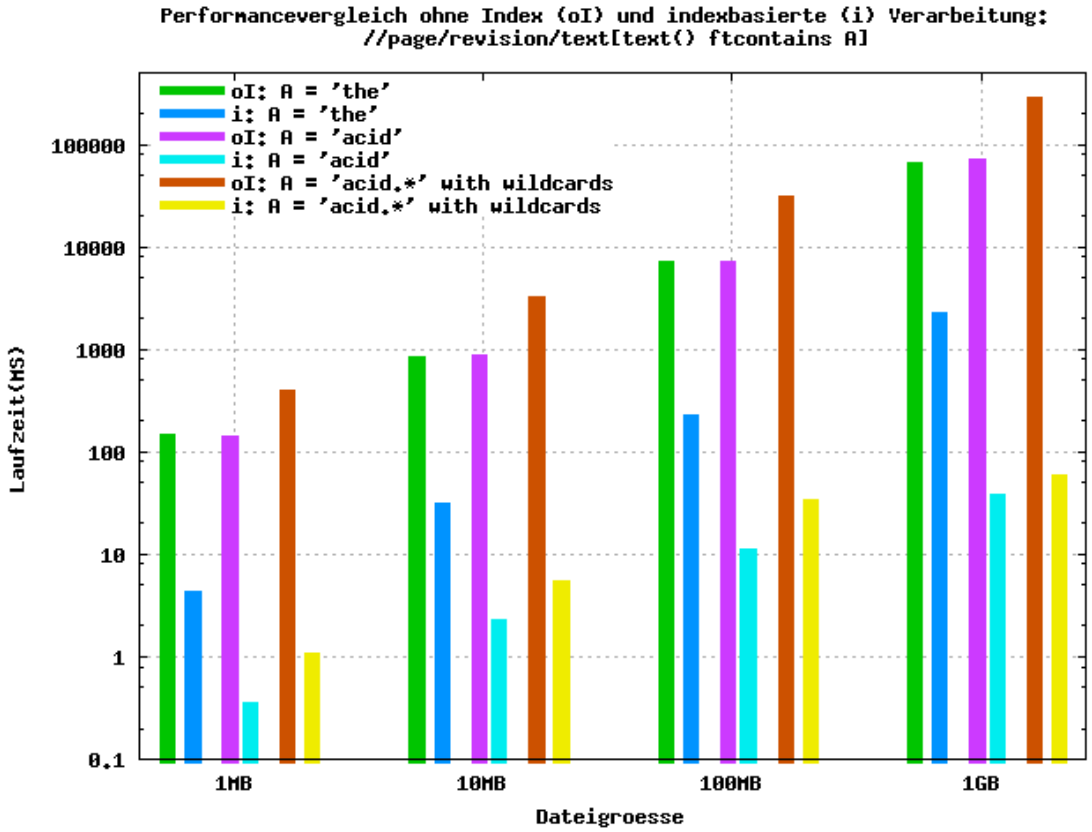


Tabelle 4.2: Anzahl Treffer

	1 MB	*10 MB*	*100 MB*	*1 GB*
the	29	501	5959	100843
acid	2	27	301	1994
acid.*	2	30	339	2297

Zunächst einmal fällt in Abbildung (4.3) sehr deutlich der enorme Performancevorteil des indexbasierten Modus auf. Im sequentiellen Modus (oI) ist sehr gut zu erkennen, dass die Anzahl der Treffer (Tabelle (4.2)) keinen Einfluss auf die Laufzeit der Anfrage

hat, da die Höhe der Balken für "the" - häufigstes Wort und "acid" - eines der seltensten Wörter - nahezu gleich sind. Hier spielt vielmehr die Länge des Full-Textes eine Rolle. Im indexbasierten Modus hingegen ist sehr deutlich zu erkennen, dass die Anzahl der Treffer die Laufzeit der Anfragen maßgeblich beeinflusst.

Die Wildcardsuche "acid.*" liefert kaum mehr Treffer als die Suche nach "acid", allerdings ist die Performance deutlich schlechter. Der sequentielle Modus nutzt zur Wildcardsuche einen regulären Ausdruck, dessen Auswertung zusätzliche Zeit benötigt. In dem indexbasierten Modus - hier liegt der Compressed Trie zu grunde (siehe Abschnitt (3.3.2)) - hängt die Performance von Wildcardanfragen sehr stark von der Position der Wildcard ab. Je länger der Präfix vor der Wildcard ist, desto effizienter kann die Baumstruktur ausgenutzt werden. Daher kann nicht grundsätzlich davon ausgegangen werden, dass Wildcardanfragen im indexbasierten Modus eine bessere Performance haben als im sequentiellen. Wesentliche Faktoren, die hier eine Rolle spielen, sind die Position(en) und die Anzahl der Wildcard(s), sowie die zu Grunde liegende Index-Struktur.

Im Folgenden wird aufgezeigt, dass die Pfadumkehrung nicht immer möglich ist bzw. zwangsläufig zu einer effizienten Ausführungsstrategie führt, und es werden Lösungsvorschläge hierfür aufgezeigt.

4.3.2 Sequentiell indexbasierter Verarbeitungsmodus

Die in Abschnitt (4.3.1) beschriebe Pfadumkehrung ist bei Anfragen, die zwar einen Full-Text Bezug haben, jedoch den Knotentext nicht berücksichtigen, grundsätzlich nicht möglich, da die Full-Text Index-Struktur nicht angewendet werden kann.

```
//MEDIUM/LAN["dt" ftcontains "dt"]
```

In obiger Anfrage kann bereits im Compilierungsschritt erkannt werden, dass das Prädikat `["a" ftcontains "a"]` für jeden Knoten `true` ergibt und so aus dem Anfragebaum entfernt werden kann. Nun ist eine Pfadumkehrung nicht mehr möglich/nötig.

Index-Strukturen ermöglichen die effiziente Suche anhand von Suchschlüsseln. Werden jedoch alle Daten gesucht, die gerade nicht dem Suchschlüssel entsprechen, ist i.d.R. die gesamte Index-Struktur zu durchsuchen. Solche Anfragen sind mit XQuery Full-Text grundsätzlich mittels dem logischen Operator FTUnaryNot (siehe Abschnitt (2.3.3)) spezifizierbar. Der zweite Eintrag in Tabelle (4.3) liefert ein Beispiel für ein Prädikat einer solchen Anfrage. Diese Klasse von Anfragen kann über eine Erweiterung des sequentiellen Verarbeitungsmodus (siehe Abschnitt (4.2)) evaluiert werden. Die Tabelle (4.3) gibt zunächst einen Überblick über weitere dieser Fälle:

Tabelle 4.3: Übersicht Full-Text Prädikate mit FTUnaryNot und effiziente Indexnutzung

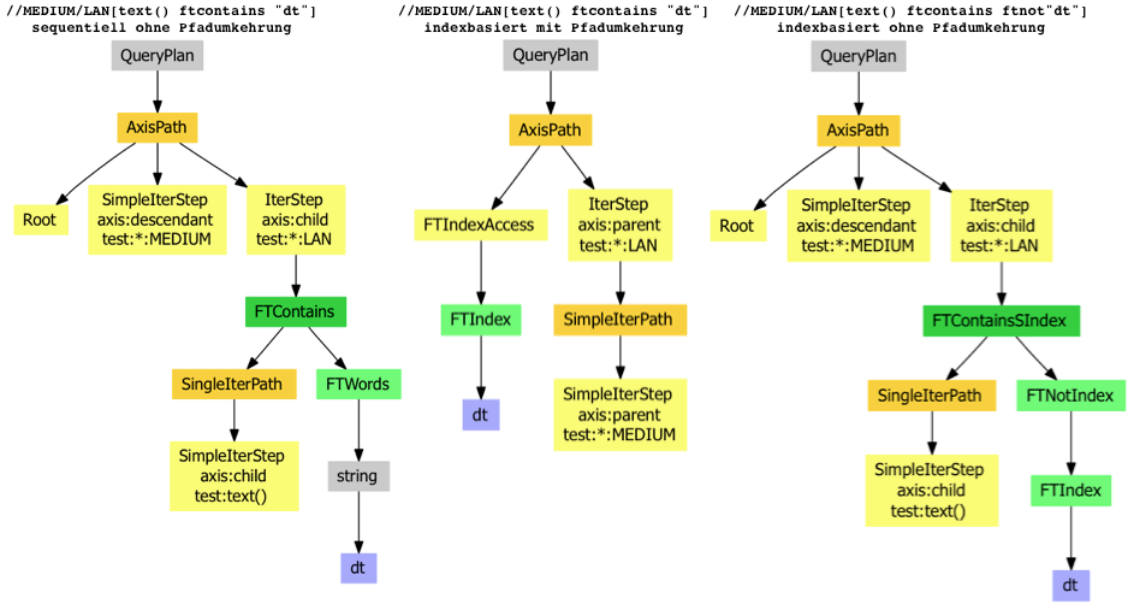
PRÄDIKAT	*EFFIZIENTE INDEXNUTZUNG*
"a" ftcontains "a"	Keine Indexnutzung, Knotentext nicht referenziert
/step$_1$/step$_2$/$\cdots$/text() ftcontains not "a"	Indexnutzung ohne Pfadumkehrung
/step$_1$/step$_2$/$\cdots$/text() ftcontains "a" ftor not "b"	Indexnutzung ohne Pfadumkehrung
/step$_1$/step$_2$/$\cdots$/text() ftcontains "a" not in not "b"	Indexnutzung ohne Pfadumkehrung
/step$_1$/step$_2$/$\cdots$/text() ftcontains "a" ftand not "b"	Indexnutzung mit Pfadumkehrung

Anfragen, die eine Indexunterstützung ohne Pfadumkehrung ermöglichen, werden sequentiell indexbasiert verarbeitet, d.h., es wird für jeden Ergebnisknoten des Pfadausdrucks das Prädikat unter Zuhilfenahme der Index-Struktur evaluiert. Anhand der Anfrage

```
//MEDIUM/LAN[text() ftcontains ftnot "dt"],
```

deren Anfrageplan der Abbildung (4.4) (rechts) zu entnehmen ist, soll das Vorgehen erläutert werden.

Abbildung 4.4: Anfragepläne sequentiell, indexbasierter und sequentiell indexbasierter Verarbeitungsmodus

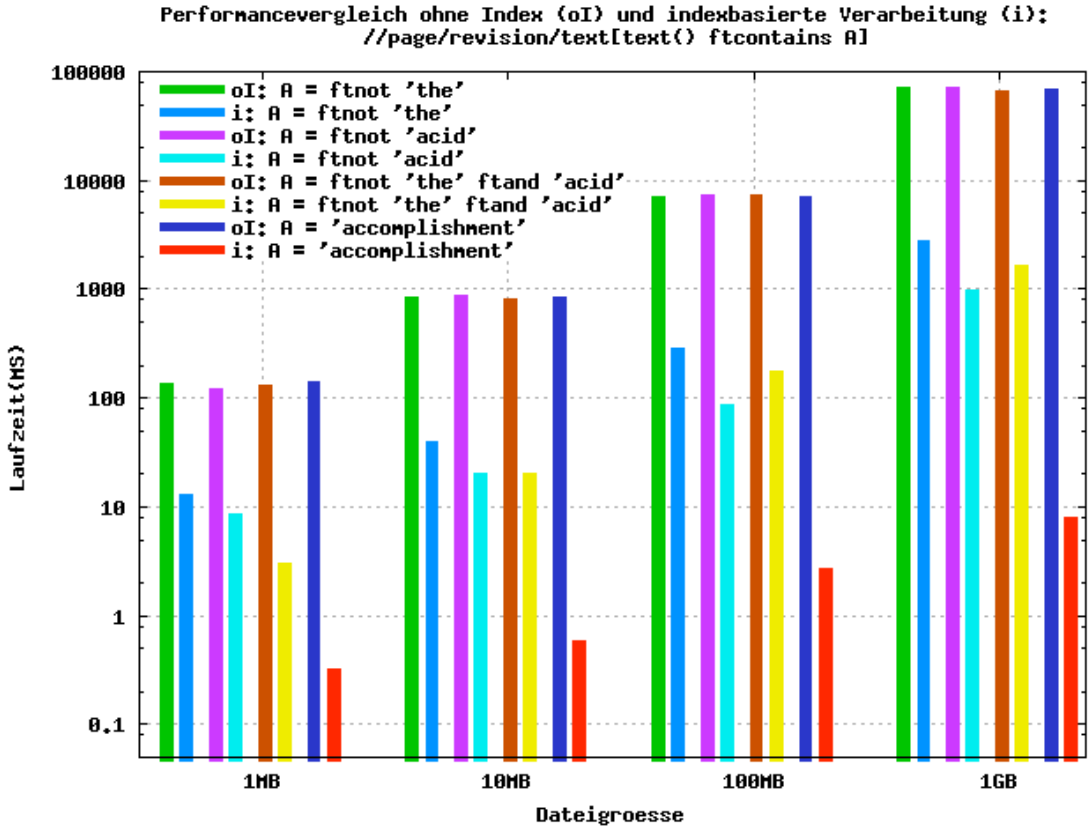


Der IterStep-Ausdruck liefert eine Knotenmenge, die dem Pfadausdruck vor dem Prädikat entspricht. Nun kann hierauf in der FTConainsSIndex Expression der Pfadausdruck in dem Prädikat geprüft werden. Die Index-Struktur liefert nun der FTContainsSIndex Expression alle Treffer zu dem Token "dt" und die FTContainsSIndex Expression vereinigt diese beiden Treffermengen, indem die Knotenmenge um die Indextreffer zu dem Token "dt"bereinigt werden. Somit kann der Index auch für solche Anfragen effizient genutzt werden. Die Verarbeitung komplexerer Anfragen wird in Abschnitt (5) detailliert beschrieben.

Die Abbildung (4.4) zeigt weiterhin beispielhafte Anfragepläne der verschiedenen Verarbeitungsmodi. Die beiden Anfragepläne (links, rechts) im sequentiellen Modus zeigen sehr deutlich, dass zunächst der Pfadausdruck vor dem Prädikat ausgewertet wird. Dem entgegen stellt der Anfrageplan des indexbasierten Modus die Auswertung des Pfadausdrucks voran.

Bei der Performance kann auch hier wieder ein klarer Vorteil für die indexbasierte Verarbeitung festgehalten werden. In der Abbildung (4.5) ist sehr deutlich zu erkennen, dass die sequentiell indexbasierte Verarbeitung (i) sehr viel effizienter ist als die sequentielle ohne Index (oI). Wie auch in Abbildung (4.3) beeinflusst die Full-Textlänge maßgeblich die Laufzeit im sequentiellen Modus.

Abbildung 4.5: Performancevergleich sequentielle und indexbasierte Verarbeitung



Die Performance des sequentiell indexbasierten Modus (i) lässt sich wie folgt erklären: Bei der Anfrage

```
//page/text[text() ftcontains ftnot 'the']
```

liefert die FTContainsSIndex Expression (analog zu Abbildung (4.4)) sehr viele mehr Textknoten als die beiden anderen Anfragen, daher benötigt diese am meisten Zeit. Der

Aufwand für die voranstehende Pfadprüfung ist für alle drei Anfragen gleich. Interessant ist die nahezu gleiche Balkenhöhe der beiden Anfragen bei ca. 10 MB Dateigröße. Bis zu dieser Grenze ist die Anfrage

`//page/text[text() ftcontains ftnot 'the' ftand 'acid']` schneller als
`//page/text[text() ftcontains ftnot 'acid']`,

obwohl in der ersteren Anfrage noch ein FTAnd berechnet werden muss. Da aber `'acid'` nur sehr selten vorkommen, in dem 10 MB Dokument 27 mal, `'the'` jedoch 501 mal, liefert `ftcontains ftnot 'the' ftand 'acid'` überhaupt keine Treffer. Damit sind alle Knoten, die dem Pfadausdruck vor dem Prädikat entsprechen, Ergebnisknoten. Für größere Dokumente entspricht das Anfrageverhalten den Erwartungen.

Der Kompilierungsschritt (siehe Abschnitt (4.1)) kann für Full-Text Anfragen erweitert werden, um den passenden Verarbeitungsmodus auszuwählen. Hierbei ist der Full-Text Teilbaum (ab der FTContains Expression in Abbildung (4.4)) des Iterator Tree zu traversieren und nach FTUnaryNot-Expression zu durchsuchen, die einer Pfadumkehrung im Wege stehen. Die gesammelten Informationen ermöglichen dann im Kompilierungsschritt der AxisPath Expression die Auswahl des entsprechenden Verarbeitungsmodus.

4.3.3 Grenzen und alternative Konzepte zur Indexanbindung

Das in dieser Arbeit vorgestellte Konzept zur Indexanbindung stellt sicher, dass Index-Strukturen immer dann verwendet werden, wenn ihr Einsatz möglich und sinnvoll ist. Es ist völlig autonom und kommt ohne durch den Anwender eingebrachte Zusatzinformationen aus.[1]

Alternativ wäre die Einführung neuer Full-Text Operatoren möglich, deren Evaluation ausschließlich an eine Indexnutzung gebunden ist, z.B. eingesetzt in eXist [19] oder Sedna [11]. Allerdings muss der Anwender dann explizit diese Operatoren in Anfragen verwenden und der XQuery-Standard wird außer Acht gelassen, so dass eine Portierung der Anfragen auf andere Systeme nicht mehr ohne Anpassungsaufwand möglich ist. Fraglich ist auch, wie sich das System verhält, wenn Anfragen nicht über den Index beantwortet werden können, er jedoch referenziert wird.

Liegt eine XML-Schema Definition vor, kann für darin definierte Datentypen auch die Verwendung einer Index-Struktur zur Anfrageauswertung gesteuert werden. Ebenso können fixe Pfade definiert werden, deren Blattelemente indexiert werden. Verwenden Anfragen jedoch Elemente, die nicht auf diesem Pfad liegen, kann die Index-Struktur nicht verwendet werden.

Der hier gefundene Ansatz ist allgemein gültig und kommt darüber hinaus ohne Zusatzinformationen aus. Er ist unabhängig von vorher zu definierenden Pfaden oder Schemainformationen und führt Indexzugriffe ohne Nutzerinteraktion vollkommen automatisch

1 Indizes können in BaseX auch explizit angesprochen werden: `index("keyword", "indextype")`

aus. Allerdings kann die Index-Struktur nur verwendet werden, wenn der Full-Text in der Form gespeichert ist, die auch die Anfrage voraussetzt. Spezifiziert eine Anfrage case sensitive Suche, die Index-Struktur hat jedoch den Full-Text case insensitive gespeichert, dann muss die Index-Struktur eine entsprechende Funktionalität bereit stellen, die die case sensitive Suche auf einer insensitiven Struktur ermöglicht, oder sie kann nicht verwendet werden. Diese speziellen Anfragen ermöglichen die FTOptions (siehe Abschnitt (2.6)). Hier erscheint es ratsam, mehrere speziell optimierte Indexstrukturen einzusetzen, um den vollen Funktionsumfang effizient zu unterstützen. Alternativ ist der sequentielle Ansatz zu wählen, der die volle Funktionalität bietet. Allerdings sind bei dieser Wahl auch Effizienzgesichtspunkte zu berücksichtigen, die in Abschnitt (6.2) analysiert werden.

KAPITEL 5

Integration der XML-Full-Text Funktionalität

5.1 Integration iterativer Full-Text Operatoren

Im voranstehenden Kapitel ist bereits der Weg vom Expression Tree zum Iterator Tree bei der Anfrageverarbeitung, unter Berücksichtigung von vorhandenen Index-Strukturen, beschrieben worden. In diesem Kapitel wird aufgezeigt, wie eine XQuery Full-Text Implementierung unter Anwendung des allgemeinen Iterator Konzeptes aussehen kann - mit dem Focus auf Nutzung einer Full-Text Index-Struktur.

Die konsequente Fortsetzung des Iteratorkonzeptes für XQuery Full-Text ermöglicht die absolute Kompatibilität zur vorhanden XQuery Implementierung und eröffnet Optimierungspotential. Darüber hinaus ist eine mengenbasierte Implementierung sehr viel aufwändiger, da diese Mengen von Knoten und Full-Text Daten verarbeitet. Zusätzlich entspricht dieses Konzept nicht dem iterativen XQuery Ansatz. Somit wertet der gesamte Teilbaum unter dem FTContains Iterator in einem Durchlauf auch nur einen Textknoten aus. D.h., der Index liefert den nächsten Full-Text Treffer, dieser durchläuft den Teilbaum bis der FTContains Iterator erreicht ist. Falls dieser Knoten einen der logischen Operatoren bzw. die FTMatchOptions nicht erfüllt, fordert dieser den nächsten Knoten von seinem Operand an. Dieses Vorgehen setzt sich bis zum Index fort. Die Vorgehensweise ist stark an den sequentiellen Verarbeitungsmodus angelehnt, allerdings werden hier Scoring-Werte zwischen den Iteratoren ausgetauscht und keine Knoten (die auch einen Scoring-Wert enthalten).

Hierbei werden die FTMatchOptions von einem zentralen Container (FTOptions Iterator), der sowohl im indexbasierten, als auch im sequentiellen Verarbeitungsmodus referenziert wird, ausgewertet, so dass sichergestellt ist, dass für die gleiche Anfrage in beiden Modi die gleichen Treffer evaluiert werden. Es unterscheiden sich lediglich die Datenquellen und die Implementierung der logischen Full-Text Operatoren. Dieser Container ist in der Arbeitsgruppe entwickelt worden und soll in dieser Arbeit nicht weiter thematisiert werden.

5.2 Implementierung iterativer Full-Text Operatoren

5.2.1 Sequentieller Verarbeitungsmodus

Im sequentiellen Verarbeitungsmodus tauschen die Iteratoren Scoring-Werte aus. Die XQuery Spezifikation des W3C gibt für Scoring-Werte einen Wertebereich von 0 bis 1 vor, wobei das eingesetzte Scoringmodell implementationsabhängig ist. Daher ist es ausreichend, die Scoring-Werte der mit einem Iterator in Verbindung stehenden Operanden zu vereinigen und zurück zu geben. Die logischen Full-Text Operatoren können wie folgt implementiert werden:

FTAnd

Alle Rückgabewerte der Operanden eines FTAnd Iterators werden über ein Scoringmodell konjugiert. Sollte einer der Operanden einen Scoring-Wert zurückliefern, der aussagt, dass kein Treffer gefunden wurde, kann dieser Wert zurückgegeben werden.

FTOr

Alle Rückgabewerte der Operanden eines FTOr Iterators werden über eine Oder-Verknüpfung durch das Scoringmodell zu einem Scoring-Wert zusammengefasst. Nur wenn alle Operanden einen Scoring-Wert zurückliefern, der aussagt, dass kein Treffer gefunden wurde, kann dieser Wert zurückgegeben werden.

FTMildNot

Sollte der Scoring-Wert, den der erste Operand liefert, aussagen, dass kein Treffer gefunden wurde, so ist dieser zurück zu geben. Liefert einer der nachfolgenden Operanden einen Wert zurück, der auf einen Treffer hindeutet, so muss der FTMildnot Iterator einen Scoring-Wert zurückliefern, der auf keinen Treffer hindeutet. Wurde hingegen von allen Operanden ein Scoring-Wert zurückgeliefert, der auf keinen Treffer hindeutet, so ist der Scoring-Wert des ersten Operanden zurück zu geben.

FTNot

Da eine FTNot Expression aufgrund der XQuery Grammatik (Abbildung (4.1)) nur einen Operanden haben kann, ist der Scoring-Wert des Operanden invertiert zurück zu geben. D.h., deutet der Scoring-Wert auf einen Treffer hin, muss der Rückgabe Scoring-Wert auf keinen Treffer hindeuten und umgekehrt.

FTContains

Der FTContains Iterator hat immer zwei Operanden, einen der die Expression vor dem Full-Text Ausdruck fasst, und einen für den Full-Text Ausdruck des Prädikats (z. B. FTSelect Iterator). Bei der Auswertung der FTContains Expression wird nun jedes Ergebnis des ersten Operanden ausgelesen, entsprechend der Kontext gesetzt und der jeweils resultierende Scoring-Wert der FTSelect Iteratoren konjugiert (Scoringmodell). Der über alle Einträge des ersten Operanden konjugierte Scoring-Wert wird zurückgegeben.

Dieser Operand kann sowohl einen Pfadausdruck repräsentieren, der nur einen Knoten als Ergebnis liefert, wie auch eine Sequenz von Knoten.

5.2.2 Indexbasierter Verarbeitungsmodus

Die Implementierung der logischen Full-Text Operatoren als Iteratoren im indexbasierten Verarbeitungsmodus ist komplexer als die zuvor für den sequentiellen Modus gezeigte, da sichergestellt werden muss, dass im gesamte Full-Text Teilbaum in einer Iteration auch nur auf einem Knoten gearbeitet wird. Zusätzlich müssen die Full-Text Daten des Knotens gegebenenfalls aktualisiert werden. Im Iterator Tree werden die logischen Operatoren durch ihre speziell für den indexbasierten Verarbeitungsmodus optimierten Varianten ersetzt.

FTAnd - FTIntersection

Zunächst einmal müssen zwei grundlegende Fälle bei der Betrachtung von FTIntersection (Indexvariante von FTAnd) unterschieden werden. Zum einen den, wenn es mindestens einen FTUnaryNot Iterator als Operand gibt. Im zweiten Fall gibt es ausschließlich oder überhaupt keine FTUnaryNot Iteratoren als Operanden. Im letzteren Fall reicht es aus, denjenigen Knoten zu finden, der in allen Iteratoren vorkommt, und deren Full-Text Daten zu vereinigen. Hierbei kann die Sortierung der Knoten ausgenutzt werden. Liefert einer der Iteratoren keinen Knoten mehr, kann die Verarbeitung abgebrochen werden. So bleibt das iterative Konzept in dem Full-Text Operator erhalten.

Gibt es sowohl Operanden die FTUnaryNot Iteratoren sind, wie auch Operanden, die keine FTUnaryNot Iteratoren sind, so kann das obige Verfahren zunächst für beide Gruppen getrennt ausgeführt werden, um dann die beiden Knoten mit anderen zu verglichen. Ist die ID des nicht aus den FTUnaryNot Iteratoren stammenden Knotens kleiner, kann dieser als Treffer zurückgegeben werden. Sind die IDs gleich, so muss das Verfahren von vorne beginnen, und ist die ID größer, so müssen die FTUnaryNot Iteratoren nach dem nächsten gemeinsamen Knoten suchen und den ID-Vergleich wiederholen. Liefert einer der nicht FTUnaryNot Operanden keinen gemeinsamen Knoten mehr, so kann die Verarbeitung abgebrochen werden. Auch hier bleibt das iterative Konzept erhalten.

FTOr - FTUnion

Bei dem FTOr Operator kann es vorkommen, dass FTUnaryNot Iteratoren als Operanden an dem FTOr anhängen. In diesem Fall wird im Compilierungsschritt die De Morgan'sche Regel angewandt.

$$\neg A \ \vee \ \neg B \ = \ \neg(A \ \wedge \ B) \ \Rightarrow \ ftnot \ A \ ftor \ ftnot \ B \ = \ ftnot(A \ ftand \ B)$$

Der Vorteil dieser Umformung besteht darin, dass die FTUnion Expression (Indexvariante von FTOr) so maximal einen FTUnaryNot Iterator als Operand haben kann. Damit wird die Implementierung weitestgehend vereinfacht.[1] Nun kann von allen Operanden derjenige mit der kleinesten ID ermittelt und zurückgegeben werden. Sollten Knoten die gleiche

1 Da FTOr Expressions kommutativ sind, kann deren Reihenfolge beliebig geändert werden.

ID haben, müssen deren Full-Text Daten vereinigt werden. Allerdings müssen die nicht zurückgegebenen Knoten zwischengespeichert werden, da ein Iterator jeden Knoten nur ein einziges Mal zurück gibt, d.h. beim nächsten Aufruf der Iterator auch den nächsten Knoten zurückgibt.

Knoten, die aus einem FTUnaryNot Iterator stammen, tragen ein besonderes Merkmal, so dass sie von dem FTContains Iterator erkannt und entsprechend verarbeitet werden können.

FTMildNot - FTMildNotIndex

Sofern nur der erste Operand Knoten als Treffer zurückliefert, können diese von dem FTMildNotIndex Iterator zurückgegeben werden. Sonst ist aus den übrigen Operanden derjenige Knoten mit der kleinsten ID zu ermitteln und mit der ID des Knotens aus dem ersten Operand zu vergleichen. Sollte die ID aus dem ersten Operand kleiner sein, kann dieser Knoten zurück gegebenwerden. Sind hingegen beide IDs gleich, müssen die Full-Text Daten herangezogen werden, um die "Mild Not"- Bedingung zu prüfen. Ist dagegen die kleinste ID der übrigen Operanden kleiner als die ID des ersten Operand, so ist obige Prozedur mit der nächst größeren ID aus den übrigen Operanden fortzusetzen.

FTUnaryNot - FTUnaryNotIndex

Jeder Knoten, der in dem Full-Text Teilbaum von den Iteratoren verarbeitet wird, trägt ein besonderes Merkmal (Negativmerkmal), so dass die Iteratoren Knoten erkennen können, die nicht in der Ergebnismenge auftreten dürfen (siehe Abschnitt (4.3.2)). Der FTUnaryNotIndex Iterator hat analog zu dem FTUnaryNot Operator nur einen Operanden. Bei dem von ihm zurück gegebenen Knoten wird das Negativmerkmal invertiert und der Knoten zurückgegeben.

FTContains - FTContainsIndex

Der FTContains Iterator wird im indexbasierten Verarbeitungsmodus mit Pfadumkehrung (siehe Abschnitt (4.3.1)) verwendet. Im Kompilierungsschritt wird aus der FTContains Expression ein FTContainsIndex Iterator, sofern eine Pfadumkehrung möglich ist. Der FTContainsIndex Iterator hat im Vergleich zu der sequentiellen Variante nur einen Operanden, dessen Ergebnisse an den Vater-Iterator zurückgegeben werden. Der Vater-Iterator ist immer eine AxisPath-Iterator und übernimmt die weitere Auswertung der Anfrage (vgl. Abbildung (4.2)).[1]

FTContains - FTContainsSIndex

Der FTContainsSIndex Iterator findet im sequentiell indexbasierten Verarbeitungsmodus Anwendung. Im Kompilierungsschritt wird, sofern keine Pfadumkehrung (siehe Abschnitt (4.3.2)) möglich ist, aus der FTContains Expression ein FTContainsSIndex Iterator. Dieser

1 In diesem Modus kann der Textknoten Test ignoriert werden, da die Index-Struktur nur Textknoten enthält.

Iterator hat, wie der rein sequentielle FTContains Iterator auch, immer zwei Operanden: einen der die Expression vor dem Full-Text Ausdruck fasst, und einen weiteren für den Full-Text Ausdruck des Prädikats (z. B. FTSelect Iterator). Nun werden für jeden Knoten, den der erste Iterator liefert, alle Knoten mit einer kleineren ID in dem zweiten Iterator übersprungen. Sollten nun beide IDs gleich sein, hängt es von dem Negativmerkmal des Knotens aus dem zweiten Iterator ab, welcher Wert zurückgegeben wird. Ist das Negativmerkmal gesetzt, wird ein Scoring-Wert zurückgegeben, der aussagt, das dieser Knoten kein Treffer ist. Sollte es nicht gesetzte sein, wird entsprechend ein Scoring-Wert zurückgegeben, der einem Treffer entspricht.

Sollten die IDs ungleich sein und der Knoten ein gesetztes Negativmerkmal haben, wurde ein Treffer gefunden und es wird ein scoring Wert zurück gegeben, der dies auch aussagt. Analog wird bei einem nicht gesetzten Negativmerkmal eine scoring Wert zurück gegeben, der auf keinen gefundenen Treffer hinweist.

5.3 Iterative Auswertung von Anfragen mit Indexunterstützung

5.3.1 Vorbemerkung iterative Auswertung

Im letzten Abschnitt ist die Implementierung iterativer Full-Text Operatoren gezeigt worden. Damit runden sie die iterative XQuery Implementierung ab. Allerdings bedeutet dies nicht, dass jeder Iterator auch intern real iterativ arbeitet. Es ist sehr wohl möglich, dass zunächst alle Ergebnisse gesammelt und dann erst iterativ zurückgegeben werden. Eine real iterative Verarbeitung würde bei jedem Aufruf das nächste Ergebnis neu evaluieren, anstelle sie vorher zu evaluieren und dann zwischenzuspeichern. Allerdings gibt es Iteratoren, die nicht real iterativ arbeiten können, z.B., wenn eine Sequenz sortiert werden muss. Hier kann erst sicher eine Aussage getroffen werden, wenn der Iterator alle Werte der Sequenz kennt, somit ist eine real iterative Verarbeitung nicht möglich.

Diese Unterscheidung kann auch bei dem Auslesen der Full-Text Daten aus einer Index-Struktur getroffen werden. Dies soll anhand des folgenden Beispiels beschrieben werden.

```
<d>
   <w>a b c</w>
   <w>a</w>
   <w>b</w>
   <w>c</w>
   <w>aa</w>
   <w>ab</w>
   <w>ac</w>
</d>
```

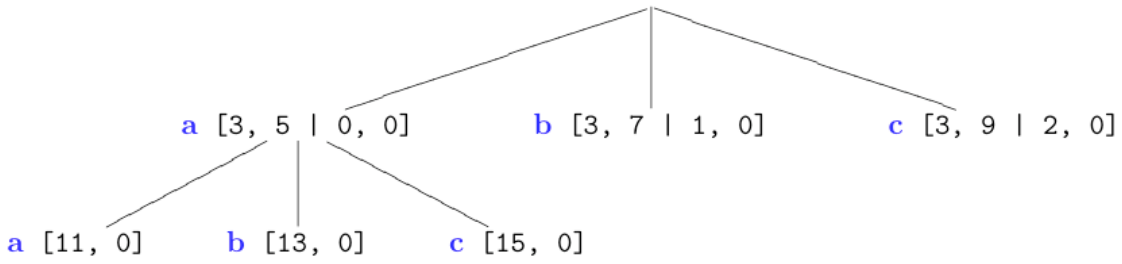
a1: `//*[text() ftcontains "a"]`

a2: `//*[text() ftcontains "a.*" with wildcards]`

Tabelle 5.1: Resultierende Datenbanktabelle

PRE	DIST	SIZE	KIND	CONTENT
0	1	16	DOC	doc.xml
1	1	15	ELEM	d
2	1	2	ELEM	w
3	1	1	TEXT	a b c
4	3	2	ELEM	w
5	1	1	TEXT	a
6	5	2	ELEM	w
7	1	1	TEXT	b
8	7	2	ELEM	w
9	1	1	TEXT	c
10	9	2	ELEM	w
11	1	1	TEXT	aa
12	11	2	ELEM	w
13	1	1	TEXT	ab
14	13	2	ELEM	w
15	1	1	TEXT	ac

Abbildung 5.1: Resultierender Full-Text Trie des Dokumentenfragments; Beschriftungsformat: **token** [pre_0, ..., pre_n | pos_0, ..., pos_n]



5.3.2 Indexstruktur ohne iterative Full-Text Datenhaltung

Bei einer Indexstruktur ohne iterative Full-Text Datenhaltung, hier ein Compressed Trie, dargestellt in Abbildung (5.1), wird bei der Suche nach "a" (Anfrage **a1**) der Baum traversiert und es werden alle Full-Text Daten die in dem Knoten mit dem entsprechenden Token "a" gespeichert sind, ausgelesen und zwischengespeichert (siehe Abbildung (5.2)).

Diese werden dann iterativ (Textknoten für Textknoten) zurückgegeben. Analog wird für Anfrage **a2** vorgegangen (Abbildung (5.3)). Hier werden nicht nur die Full-Text Daten zu dem Token "a" `[3, 5 | 0, 0]` zwischengespeichert, sondern auch noch weitergehend `[11 | 0]`, `[13 | 0]` und `[15 | 0]`, da diese auch der Anfrage **a2** entsprechen.

Abbildung 5.2: Auslesen der Full-Text Daten für die Anfrage **a1**

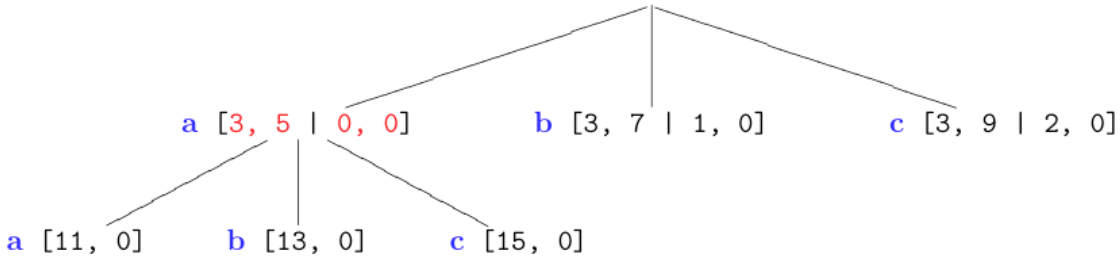
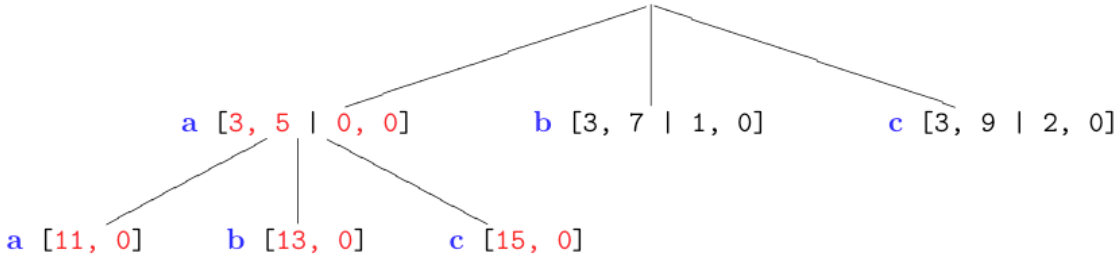


Abbildung 5.3: Auslesen der Full-Text Daten für die Anfrage **a2**



Die gesammelten Full-Text Daten werden wiederum iterativ (Knoten für Knoten) zurückgegeben. Bei Anfragen, die nur die ersten k Treffer referenzieren, kann der Fall auftreten, dass so nicht benötigte Daten eingelesen werden.

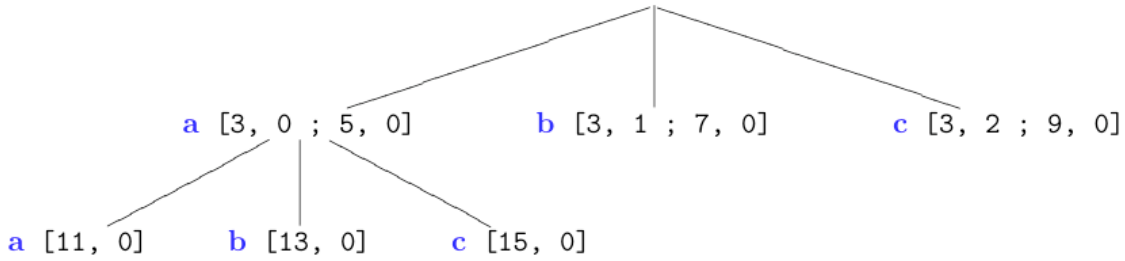
5.3.3 Indexstruktur mit iterativer Full-Text Datenhaltung

Das iterative Auslesen von Full-Text Daten aus einer Index-Struktur setzt voraus, dass die Daten auch in einer entsprechenden Form gespeichert werden. Bei der in Abschnitt (3.3.2) vorgestellten klassischen Datenhaltung wird zunächst eine Liste aller pre-Werte und dem folgend aller pos-Werte gespeichert, wobei es zu jedem pre-Wert einen pos-Wert gibt.[1] Das iterative Auslesen des nächsten pre/pos-Werte-Paares kann bei diesem Design nicht effizient durchgeführt werden, da zwischen dem pre-Wert und dem pos-Wert genau so viele Werte übersprungen werden müssen, wie es insgesamt pre-Werte gibt. Daher sind die Full-Text Daten als Folge von pre/pos-Werte-Paaren zu speichern. Die Abbildung (5.4) zeigt dieses veränderte Design, bei der jedes pre/pos-Werte-Paar durch ein ";" voneinander getrennt ist.

Diese Art der Speicherung kann auch beim nicht iterativen Auslesen der Werte genutzt werden, allerdings hat sie den Nachteil, dass bei Anfragen, bei deren Verarbeitung die pos-Werte nicht benötigt werden, sie nicht mehr völlig ignoriert werden können. Allerdings

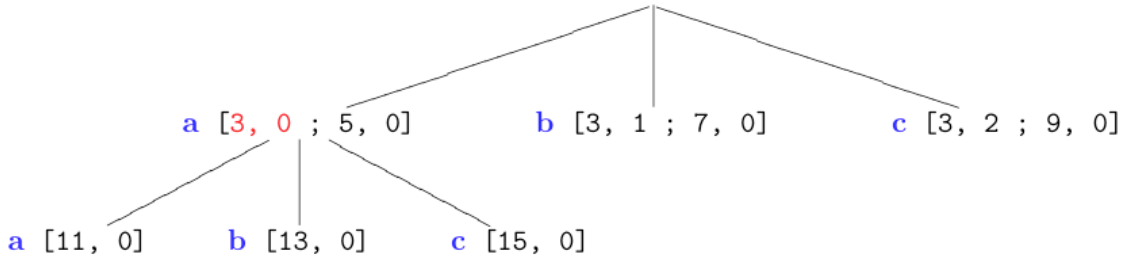
1 Der pre-Wert identifiziert den Textknoten, in dem das Token vorkommt, der pos-Wert gibt die relative Tokenposition in dem Text an.

Abbildung 5.4: Full-Text Trie mit angepasster iterativer Full-Text-Datenhaltung; Beschriftungsformat: **token** [pre_0, pos_0 ; ... ; pre_n , pos_n]



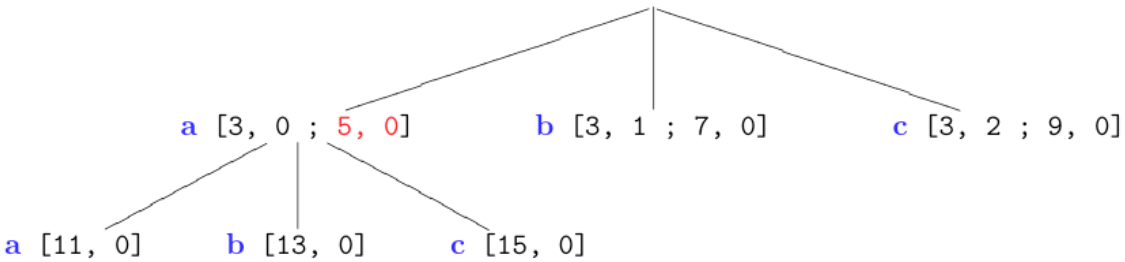
findet in der Regel eine Visualisierung der Treffer statt, bei der die pos-Werte benötigt werden, und so entfällt dieser Optimierungsschritt.

Abbildung 5.5: Iteratives Auslesen der Full-Text Daten für die Anfrage **a1** - erste Iteration.



Bei dem ersten Iterationsschritt zur Anfrageevaluation von Anfage **a1** werden nun der pre-Wert 3 und der pos-Wert 0 ausgelesen und von dem Index-Iterator zurückgegeben (Abbildung (5.5)).

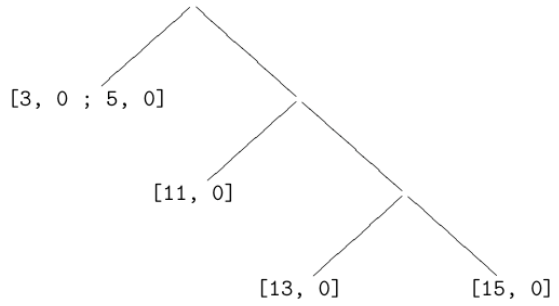
Abbildung 5.6: Iteratives Auslesen der Full-Text Daten für die Anfrage **a1** - zweite Iteration.



Im nächsten Iterationsschritt werde nun die Werte 5 und 0 ausgelesen und zurückgegeben (Abbildung (5.6)). Damit sind alle Werte aus der Index-Struktur zur Anfrage **a1** verarbeitet. Bei der Evaluation der Anfrage **a2** wird während der Wildcardsuche die Index-Struktur traversiert und für jeden Knoten der Index-Struktur, der ein Treffer ist, ein Index-Iterator erzeugt und mit den schon gefundenen verkettet. Der Index-Iterator liest zu diesem Zeitpunkt noch keine Full-Text-Daten von Festplatte. Er speichert lediglich

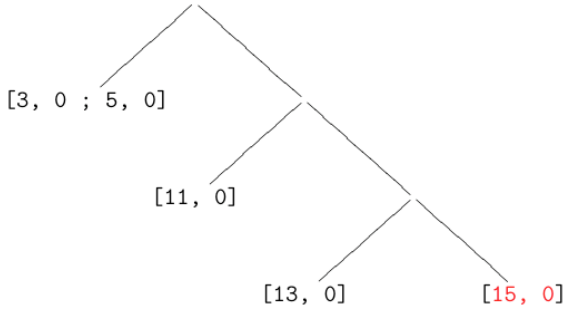
die Anzahl der zu lesenden Werte und einen Pointer auf den Startwert. Es entsteht ein Index-Iteratoren Baum, der für **a2** wie in Abbildung (5.7) gezeigt aussieht.

Abbildung 5.7: Index-Iteratoren Baum für die Anfrage **a2**.



Die Verknüpfung der Iteratoren kommt durch die Wildcard-Traversierung des Baums zustande. Wird nun der erste Treffer aus dem Index-Iteratoren Baum angefordert, so fängt der am weitesten rechts stehende Iterator an, das erste pre/pos-Werte-Paar zu lesen und gibt diesen Wert zurück (Abbildung (5.8)) - hier das Paar [15, 0].

Abbildung 5.8: Index-Iteratoren Baum: Start Suche nach minimalem pre-Wert



Nun liest sein linker Nachbar das erstes pre/pos-Werte-Paar [13, 0] ein und gibt dieses ebenfalls zurück. Der Vater Index-Iterator sucht nun den kleineren pre-Wert und gibt das entsprechende pre/pos-Werte-Paar [13, 0] zurück. Der Index-Iteratoren Baum sieht dann wie in Abbildung (5.9) gezeigt aus.

Nun liest der linke Kind Index-Iterator sein erste pre/pos-Werte-Paar und gibt den Wert an den Vater zurück. Dieser ermittelt wiederum das Minimum und gibt es zurück. Dieser Prozess setzt sich bis zur Wurzel fort und verändert den Index-Iteratoren Baum (Abbildung (5.10)).

Im nächsten Schritt ist der Wurzel bekannt, dass im gesamten Teilbaum kein kleinerer Wert mehr vorkommen kann als der pre-Wert 11, daher ist von dem linken Kind Iterator der nächste pre-Wert anzufordern. Dieser ist wiederum kleiner und wird daher zurückgegeben. Dieses Vorgehen setzt sich fort, bis der gesamte Index-Iteratoren Baum abgearbeitet ist.

Abbildung 5.9: Index-Iteratoren-Baum: Suche nach minimalem pre-Wert

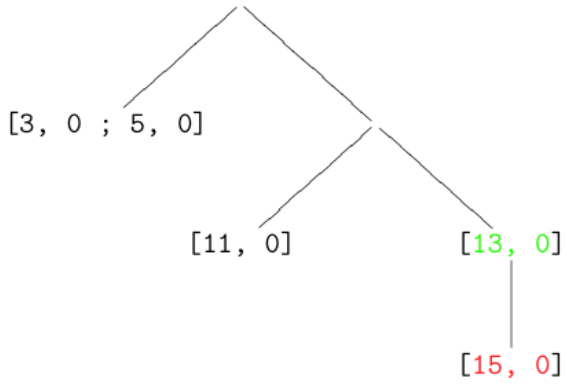
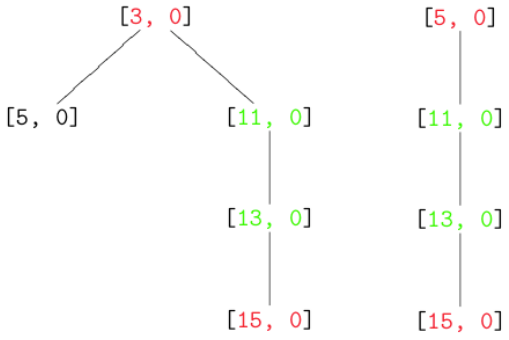


Abbildung 5.10: Index-Iteratoren-Baum: Auffinden minimaler pre-Werte



Der Aufbau von Index-Iteratoren Bäumen findet auch bei der fehlertoleranter Suche und bei unterschiedlichen Casemodi von Index-Struktur und Anfrage Anwendung.[1] Im Folgenden wird die Performance beider Varianten verglichen.

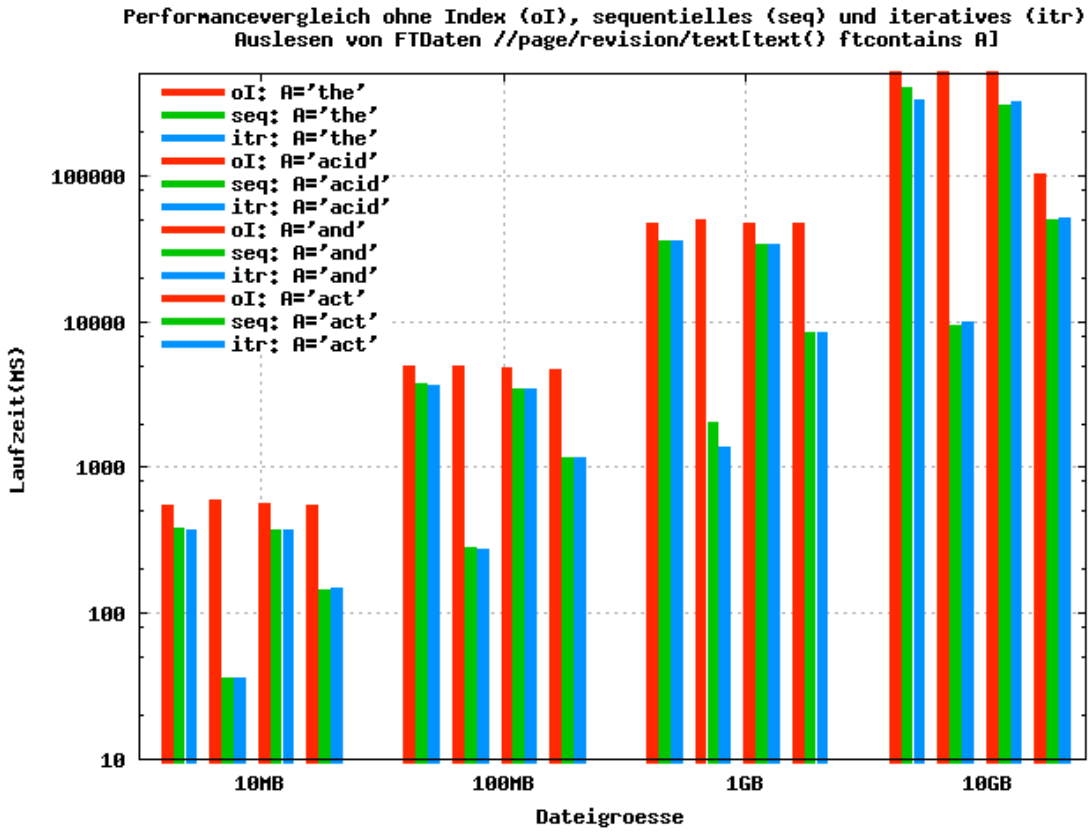
5.3.4 Performancevergleich beider Varianten

Es wird sowohl die Performance von Suchanfragen, die nur einen Suchbegriff verwenden, verglichen, wie auch Anfragen, die zwei Suchbegriffe miteinander kombinieren und in Wikipedia-Instanzen der Größe 10 MB bis 10 GB suchen.

Sowohl in Abbildung (5.11) wie auch in Abbildung (5.12) ist offensichtlich, dass die Variante ohne Index (oI) die schlechteste Performance aufweist und auch hier wieder die Laufzeit rein von der Länge der Full-Texte (Dokumentengröße) und nicht von der Anzahl der gefundenen Treffer abhängt. Die Suche nach sehr häufigen Begriffen wie "the" oder "and" benötigt erwartungsgemäß sehr viel mehr Zeit, als die Suche nach selteneren Begriffen (Häufigkeiten in Tabelle (5.2)). Dass "act" häufiger als "acid" vorkommt, aber seltener als die Stoppworte "the" und "and" ist sehr gut an der Höhe der Performancebalken zu

1 case sensitive Suche auf einer case insensitiven Index-Struktur bzw. case insensitiver Suche auf einer case sensitiven Index-Struktur

Abbildung 5.11: Performancevergleich ohne Index, sequentielles und iteratives Auslesen von FTDaten - die Anzahl der Treffer ist Tabelle (5.2) zu entnehmen



erkennen. Augenscheinlich ist auch, dass das iterative wie auch das sequentielle Auslesen der FTDaten keine signifikanten Unterschiede in der Laufzeit aufweisen.

In Abbildung (5.12) werden Anfragen mit zwei konjugierten Suchbegriffen betrachtet. Es fällt sofort auf, dass die Performance der Variante ohne Index identisch zu der Performance der Suche nach nur einem Suchbegriff ist (Abbildung (5.11)). Auffällig ist allerdings, dass die indexbasierten Varianten sehr viel schneller bzw. gleich schnell im Vergleich zu der Suche nach "the" mit nur einem Suchbegriff sind (Abbildung (5.11)). Dies erscheint zunächst wenig intuitiv, da in diesem Fall Full-Text Daten von zwei Suchbegriffen zu lesen sind und konjugiert werden müssen. Allerdings muss für jeden Full-Text Treffer auch noch der Pfadausdruck `//page/revision/text`[1] geprüft werden. Daher hat die in Tabelle (5.2) gezeigte Anzahl an Treffern einen wesentlichen Einfluss auf die Gesamtlaufzeit der Anfrage.

1 In diesem Fall wird intern der invertierte Pfad `parent::text[parent::revision/parent::page]` geprüft.

Abbildung 5.12: Performancevergleich ohne Index, sequentielles und iteratives Auslesen von FTDaten - die Anzahl der Treffer ist Tabelle (5.2) zu entnehmen

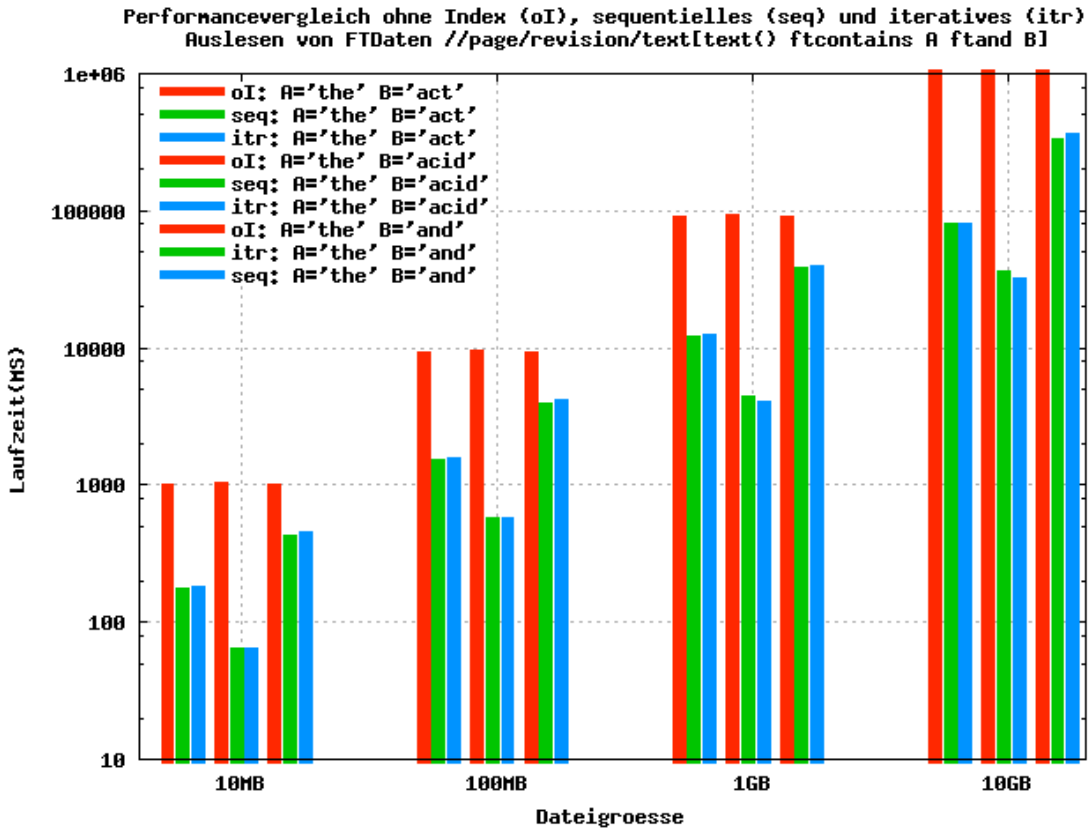


Tabelle 5.2: Übersicht Anzahl Treffer der Anfragen aus Abbildung (5.11) und Abbildung (5.12)

	"the"	*"acid"*	*"and"*	*"act"*	*"the" ftand "act"*	*"the" ftand "acid"*	*"the" ftand "and"*
10 MB	501	27	512	108	103	27	489
100 MB	5959	301	5782	1117	1095	294	5666
1 GB	100843	1994	97199	10474	10356	1962	95635
10 GB	21405210	17454	1788375	92604	89767	15516	1682015

So liefert die Suche nach `//page/revision/text[text() ftcontains "the"]` bei der 10 GB Instanz rund 21,5 Mio Treffer, hingegen `//page/revision/text[text() ftcontains "the" ftand "act"]` nur rund 89.000 Treffer. Die Pfadprüfung auf den 21,4 Mio dauert offensichtlich wesentlich länger, als die 21,5 Mio "the" Treffer mit den 92.604 "act" Treffern zu konjugieren und auf den 89.767 Treffern die Pfadprüfung durchzuführen. Analoges gilt für die Anfrage `//page/revision/text[text() ftcontains "the" ftand "acid"]`. Die Konjunktion der beiden Stoppworte "the" und "and", die für die 10 GB Instanz ca. 1,7 Mio Treffer liefert, ist immerhin noch ca. 10% schneller als die Suche nach "the". Hier scheint die Konjunktion

der 21,4 Mio "the" Treffer mit den 1,8 Mio "and" Treffern einen wesentlichen Anteil an der Gesamtlaufzeit zu haben.

Es lässt sich grundsätzlich festhalten, dass das iterative Auslesen der FT-Daten die Laufzeit von Anfragen nicht negativ beeinflusst, da keine der Anfrage in diesem Modus eine schlechtere Performance aufweist als im sequentiellen Modus. Bei der Berechnung von Konjunktionen kann sogar eine leichte Beschleunigung beobachtet werden, die bei steigender Dateigröße in Abbildung (5.12) zu erkennen ist. Diese Beschleunigung kann durch die Berechnung der Konjunktion erklärt werden: wenn eine sehr große Menge mit einer eher kleinen Menge an Knoten konjugiert werden muss und die kleine Menge komplett abgearbeitet ist, so müssen von der größeren Menge keine weiteren Werte mehr gelesen werden. Im sequentiellen Modus werden zunächst alle Werte gelesen und die Konjunktion auf den gesamten Resultatmengen gebildet. Wird in einem solchen Fall die sehr große Menge an Treffern aus einem Index-Iteratoren-Baum (siehe Abschnitt (5.3.3)) gewonnen, z.B. aufgrund einer Wildcard Anfrage, ist das Optimierungspotential noch sehr viel größer. Damit ist das Iteratoren-Prinzip konsequent bis in die Index-Struktur fortzuführen. In Abschnitt (6) werden weitere Optimierungsmöglichkeiten, die sich aus diesem Ansatz ergeben, aufgezeigt.

KAPITEL 6

Optimierungsansätze für Pfadausdrücke

6.1 Iterative Verarbeitung von Prädikaten mit Positionsfilter

Das im vorigen Kapitel beschriebene iterative Konzept und dessen konsequente Fortführung bis in die Mechanismen der Index-Strukturen kann, wie bereits angedeutet, auch zur Optimierung von Anfragen genutzt werden. Die bisherigen Betrachtungen bezogen sich rein auf die Full-Text bezogenen Operatoren und deren vollständig iterative Implementierung. Für die Optimierung ist es jedoch notwendig, dass auch der Pfadausdruck vollständig iterativ evaluiert wird. D.h., jeder Iterator des Iterator Tree einer Anfrage muss vollständig iterativ implementiert sein, da es sonst möglich ist, dass einzelne Iteratoren zunächst alle Zwischenergebnisse sammeln und diese dann erst iterativ zurückgeben. Ziel ist es, dass der Root-Iterator den nächsten Treffer anfordert und dann nur so viele Full-Text Daten von Knoten aus der Index-Struktur gelesen werden, die notwendig sind, um den nächsten Treffer zu ermitteln. Anhand der folgenden Anfrage soll das Vorgehen verdeutlicht werden.

```
/mediawiki/page/revision/text[text() ftcontains "the"]
```

Der rechte Teilbaum in Abbildung (6.1) unter dem Iterator AxisPath evaluiert den Pfad `parent::revision/parent::page/parent::mediawiki/parent::document-node()`, welcher durch die Pfadumkehrung von `/mediawiki/page/revision` zustande kommt.[1] Der AxisPath ist intern jedoch nicht vollständig iterativ implementiert, d.h., es werden intern zunächst alle Treffer, die aus dem IterStep (axis:parent, test:text) stammen, gesammelt und dann die einzelnen Parent-Step hierauf geprüft. Die vollständige iterative Variante des AxisPath ist der SimpleIterPath, der für den aktuellen Knoten die Parent-Steps prüft und das evaluierte Ergebnis zurückgibt. Diese Ersetzung kann hier stattfinden, da alle Steps des AxisPath nur Parent-Steps sind. Der Iterator-Tree in Abbildung (6.2) zeigt den vollständig iterativen Iterator-Tree.

Intern arbeitet der SimpleIterPath so, dass vom ersten SimpleIterStep der erste Knoten geholt wird, der ein revision-Element als Parent hat und auf diesem dann geprüft wird, ob dieser ein page-Element als Parent hat. Dies setzt sich fort, bis der Knoten den letzten Step

1 Diese Pfadumkehrung - eingeführt in Abschnitt (4.3.1) - ist notwendig um die Full-Text Index-Struktur verwenden zu können.

Abbildung 6.1: Iterator Tree für `/mediawiki/page/revision/text[text() ftcontains "the"]`

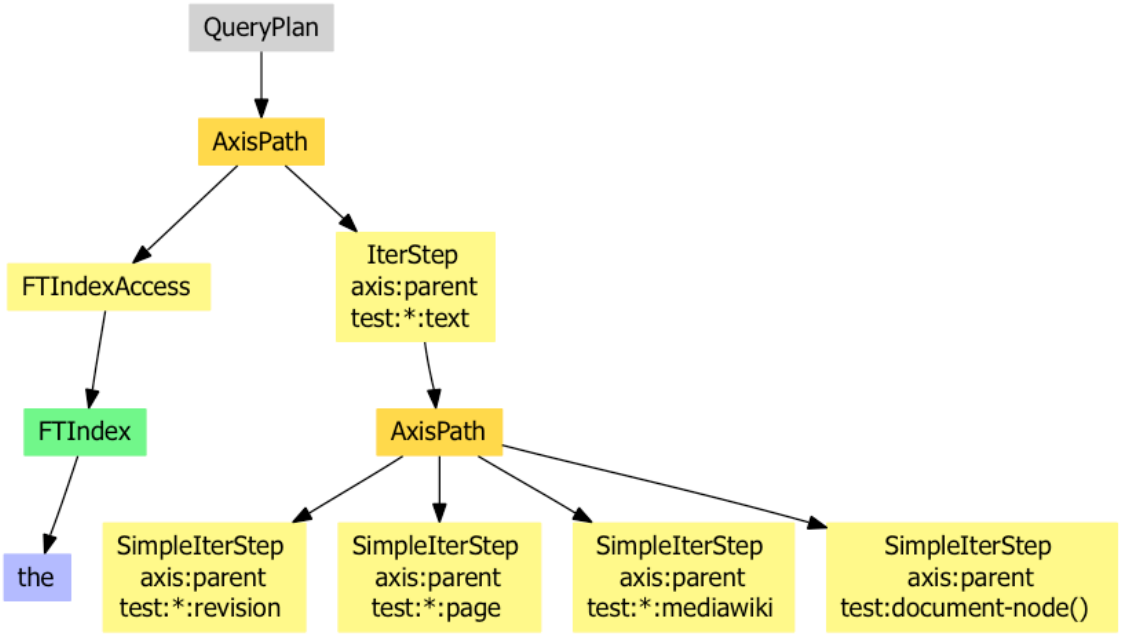
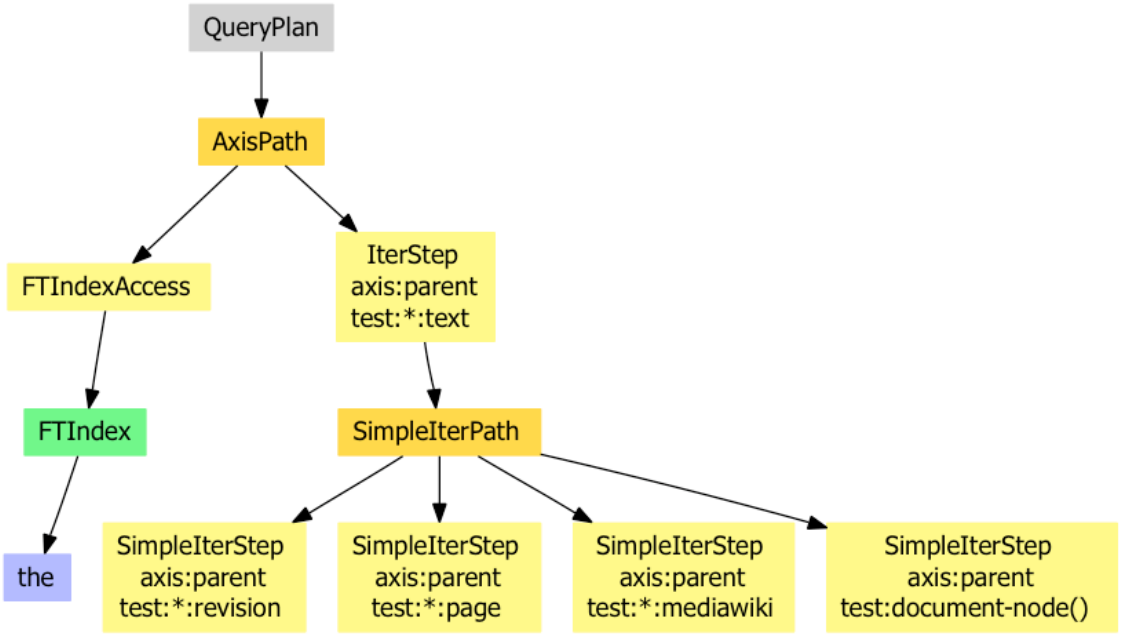


Abbildung 6.2: vollständig iterativer Iterator Tree für `/mediawiki/page/revision/text[text() ftcontains "the"]`

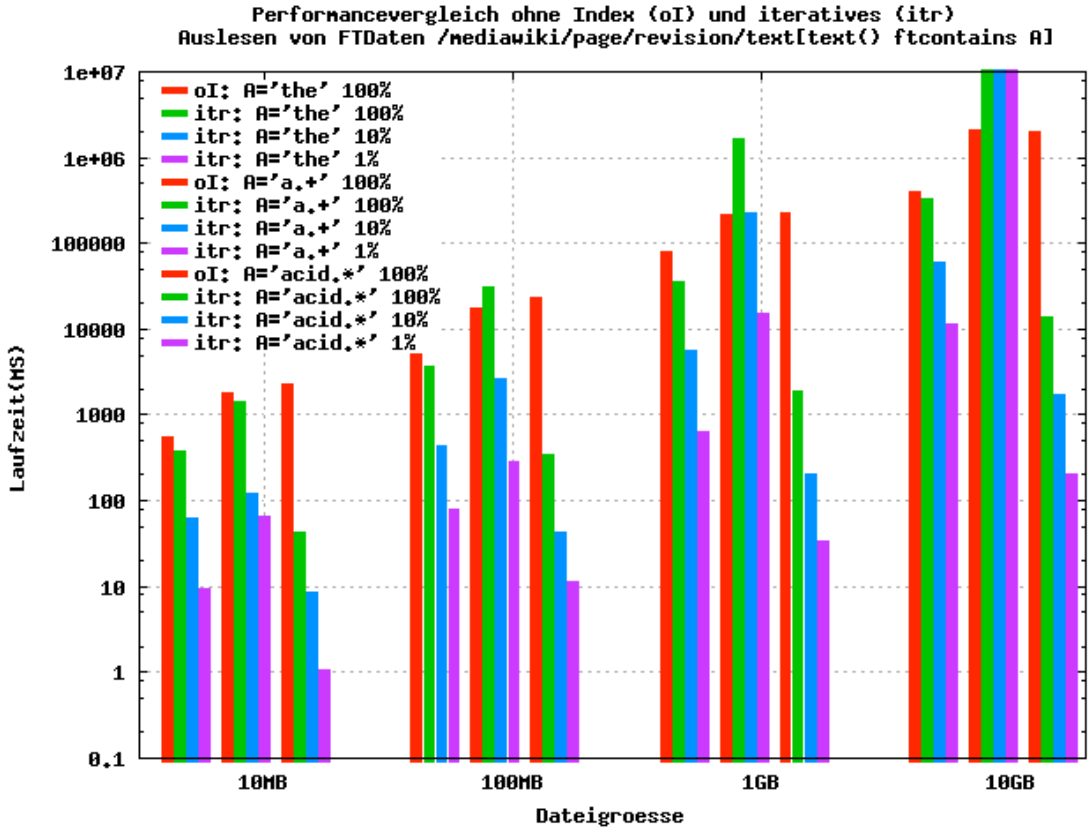


erfüllt hat. Erfüllt einer der Knoten einen Step nicht, so wird von dem davor liegenden

Step der nächste Knoten geholt und hierauf die Prüfung wiederholt. Liefert einer der Steps keine Knoten mehr, so ist die Verarbeitung abzubrechen. Sind nun alle Iteratoren des Iterator-Trees zu einer Anfrage vollständig iterativ implementiert, so kann dies zur Optimierung ausgenutzt werden.

Grundsätzlich kann die iterative Verarbeitung immer dann abgebrochen werden, wenn alle Ergebnisse der Anfrage bereits ermittelt sind. So ist es ausreichend für eine Anfrage, die nur die ersten k Knoten als Treffermenge spezifiziert, die Anfrageauswertung, nachdem der k-te Knoten als Ergebnis feststeht, vorzeitig abzubrechen. Bei einem nicht vollständig iterativen Ansatz werden zunächst alle Knoten gesammelt und von diesen die ersten k als Treffer zurückgegeben - hier ist der vorzeitige Abbruch nicht möglich. In XQuery lassen sich solche Anfragen über Positionsfilter formulieren. Allerdings gibt es auch Klassen von Anfragen, bei denen ein vorzeitiger Abbruch nicht möglich ist. Beispielhaft hierfür sind Anfragen, bei denen die Anzahl der Treffer referenziert wird. Hier müssen zunächst alle Treffer iteriert werden, um die Anzahl ermitteln zu können.

Abbildung 6.3: Performance vollständig iterativer Verarbeitung



In Abbildung (6.3) ist für die Anfrage

```
/mediawiki/page/revision/text[text() ftcontains "the"]
```

das Laufzeitverhalten unter Verwendung eines Positionsfilters visualisiert. Die Prozentzahlen geben an, nach wie viel Prozent der Treffermenge die Verarbeitung abgebrochen wurde. Die Laufzeitverkürzung ist bei 10 und 1 Prozent, im Vergleich zu 100 Prozent sehr deutlich zu erkennen. Das gleiche Verhalten ist für die Anfrage

```
/mediawiki/page/revision/text[text() ftcontains "a.+" with wildcards]
```

festzustellen. Hier greift zusätzlich das in Abschnitt (5.3.3) gezeigte Verfahren, bei dem ein Index-Iteratoren Baum erzeugt wird, um die Traversierung der Index-Struktur aufgrund der Wildcard Anfrage iterativ zu implementieren. Allerdings zeigt sich auch sehr deutlich, dass in diesem Fall der Indexzugriff für Dateigrößen ab ca. 100 MB eine längere Laufzeit aufweist als die Verarbeitung der Wildcards Anfrage ohne Indexunterstützung. Grund hierfür ist der sehr kurze Präfix vor der Wildcard und der eingesetzte Compressed Trie. Hier muss der gesamte Teilbaum unter dem Knoten mit dem Wert "a" traversiert und ein Index-Iteratoren Baum aufgebaut werden, der die Full-Text Daten iterativ verwaltet. Allerdings ist der Compressed Trie wiederum für die Anfrage mit der Wildcardsuche "acid.*" sehr viel effizienter als die sequentielle Suche ohne Index, da hier der Präfix vor der Wildcard den Suchraum sehr gut einschränkt und so die baumbasierte Struktur des Compressed Trie ausgenutzt werden kann. Auch liefert die Anfrage nur wenige Treffer (23.514 im Vergleich zu 201.612 bei "a.+" 1 GB Instanz).[1]

Es zeigt sich sehr deutlich, dass die Laufzeit einer Anfrage mit Indexunterstützung sowohl von dem Traversieren der Index-Struktur, wie auch von dem Auslesen der Full-Text Daten beeinflusst wird. Im Folgenden wird das Entscheidungsproblem sequentielle Suche ohne Indexzugriff vs. indexbasierte Evaluation aufgegriffen und aufgrund einer kostenbasierten Entscheidung die Wahl der Ausführungsstrategie getroffen.

6.2 Auswertungsstrategie für Prädikate

6.2.1 Auswertungsstrategien und Aufwandsabschätzungen

Im letzten Abschnitt hat sich gezeigt, dass es nicht zwangsläufig effizient ist, eine indexbasierte Auswertungsstrategie zu wählen, wenn dies möglich ist. Daher sind Entscheidungskriterien zu definieren, anhand derer die Wahl der effizientesten Auswertungsstrategie getroffen werden kann. In relationalen Datenbanksystemen wird oft über eine Aufwandsabschätzung versucht, den realen Aufwand zur Auswertung einer Anfrage unter Verwendung einer bestimmten Auswertungsstrategie zu schätzen. Hierbei werden Statistiken herangezogen, die Aufschluss über die entstehenden Datenmengen geben, und die Implementierungsvarianten der einzelnen Operatoren werden mit verschiedenen Kosten bewertet, um eine möglichst realistische Schätzung zu erhalten [7]. Diese Vorgehensweise kann auch für die Auswertung von Full-Text Anfragen im XML-Kontext angewandt werden.

1 Die Perfomanceschwankungen sind auf die Struktur der indexsierten 10 GB Instanzbegriffe zurück zu führen.

Bei der Verarbeitung von Pfadausdrücken mit Full-Text Bezug setzt sich die Laufzeit immer aus der Summe von Pfadnavigation und Auswertung des Full-Text Ausdrucks zusammen. Findet keine Pfadumkehrung, ausgelöst durch den Indexzugriff, statt, so ist der Aufwand für die Pfadnavigation im sequentiellen und sequentiell indexbasierten Verarbeitungsmodus gleich. Wird der Pfad jedoch invertiert, ändert sich auch der damit verbundene Auswertungsaufwand. Die Auswertung des Full-Text Ausdrucks hängt im sequentiellen Modus von der Länge des Full-Textes ab, im indexbasierten Modus hingegen von der Anzahl der zu erwartenden Treffer.

Die folgende Darstellung zeigt zunächst Möglichkeiten, den Aufwand für die Auswertung von sequentiell verarbeiteten Full-Text Anfragen zu schätzen und dem anschließend eine Abschätzung für die indexbasierte Auswertung des Full-Text Ausdrucks folgen zu lassen. Eine Abschätzung des Aufwands für die Pfadnavigation wird an dieser Stelle nicht getroffen, er wird als gleich für beide Varianten angenommen.

6.2.2 Aufwandsschätzung für Pfadausdrücke mit einem Prädikat

Wie bereits in Abschnitt (3.2.2) eingeführt, verfügen XML-Dankenbanksysteme in der Regel über einen Tag Index/Value Index, in dem alle Tags, die in einem Dokument enthalten sind, indexiert werden. Damit kann auch die Häufigkeit eines Tags im gesamten Dokument ermittelt werden. Allerdings liefert der Tag Index keine Informationen über die Struktur des Dokuments und ermöglicht daher nur eine sehr grobe Abschätzung von Aufwänden. Abhilfe schafft der Einsatz einer Path Summary (siehe Abschnitt (3.2.2)), die die Struktur des Dokuments fasst und in Kombination mit dem Tag Index/Value Index eine sehr viel genauere Abschätzung des Aufwands ermöglicht. Das folgende Beispiel verdeutlich das Vorgehen.

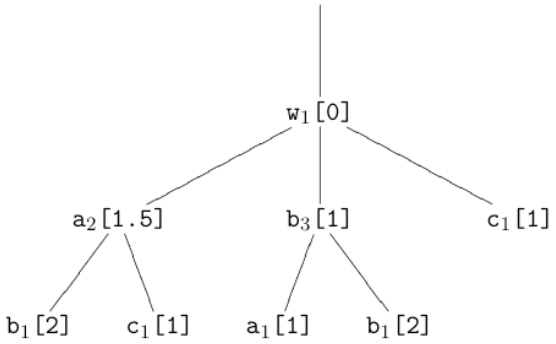
```
<w>
   <a>A A A</a>
   <b>B</b>
   <c>C</c>
   <a><b>B B</b><c>C</c></a>
   <b><a>A</a><b>B B</b></b>
   <b>B B</b>
</w>
```

Tabelle 6.1: Tag Index

TAG	PRE
b	4, 9, 13, 16, 18
a	2, 8, 14
c	6, 11
w	1

Bei der Anfrage `/w/a[b/text() ftcontains "B"]` kann aus dem Tag Index (Tabelle (6.1)) die Information gewonnen werden, dass es ein Element mit dem Tag 'w', drei Elemente mit dem Tag 'a' und fünf Elemente mit dem Tag 'b' im gesamten Dokument gibt. Somit können maximal drei Elemente Treffer dieser Anfrage sein. Aus der Path Summary (Abbildung (6.4)) ist zu entnehmen, dass es ein Element als Kind des Root-Elements mit dem Tag 'w' gibt, dieses zwei 'a' Elementkinder hat und eines der beiden ein Kindelement 'b' hat. Der Vater dieses 'a' Kindknoten ist als einziger Treffer der Anfrage `/w/a[b/text() ftcontains "B"]` zu sehen. Bei dem Evaluieren der Anfrage wird nun die Datenbanktabelle des Dokuments durchlaufen und nach Knoten gesucht, die dem spezifizierten Pfad entsprechen. Aufgrund der Information aus der Path Summary, das die Anfrage nur

Abbildung 6.4: Path Summary: t_n[k], t = Tag, n = Häufigkeit, k = ϕTextlänge



einen Treffer liefert, kann die Verarbeitung nach dem Auffinden des Treffers abgebrochen werden. Ohne diese Information müsste die gesamte Datenbanktabelle durchlaufen werden. Aufgrund der Speicherung der durchschnittlichen Textlänge (hier zwei) kann der Aufwand für eine Anfrage sehr gut abgeschätzt werden.[1]

Bei der Anfrage `//b[text() ftcontains "B"]` liefert die Path Summary eine Liste von Elementen {b_1[2], b_3[1], b_1[2]}, die wiederum zur Aufwandsabschätzung herangezogen werden kann. Ein einfaches Modell liefert, dass fünf Knoten mit einer durchschnittlichen Textlänge von 1,4 Zeichen zu durchsuchen sind.[2]

Für Anfragen, die vom root-Element aus starten, kann die Path Summary sehr effizient zur Aufwandsabschätzung für die reine Full-Text Suche genutzt werden, allerdings ist die Anwendung im Zusammenhang mit der Pfadumkehrung so einfach nicht möglich, da hier der Pfad im Blattknoten beginnt. Die Anfrage `/w/a/b[text() ftcontains "B"]` wird durch die Pfadumkehrung in `FTContainsIndex(FTIndex("B"))`
`/parent::*:b[parent::*:a/parent::*:w/parent::document-node()]` und navigiert so den Dokumentenbaum vom Blatt bis zur Wurzel.

Allerdings stehen hier weitere Informationen zur Verfügung. Sofern die Index-Struktur bei der Suche genutzt werden kann, kann die Anzahl der Treffer zu einem Suchbegriff ermittelt werden. Hierzu ist zwar ein Indexzugriff notwendig, jedoch bestimmt das Auslesen der Full-Text Daten die Laufzeit maßgeblich. Für den Suchbegriff "B" liefert die Index-Struktur sieben Treffer zurück, auf denen dann noch der invertierte Pfad zu prüfen ist. Nun ist ein Grenzwert festzulegen, der bestimmt, bis zu wie viel Prozent der Treffer eines Dokuments der Indexzugriff zu wählen ist. Hier liefert die Anfrage 100 * 7 / 13 $\approx$ 54 Prozent als Indextreffer zurück. Bei einem so hohen Wert würde die Indexanbindung sicherlich keinen Sinn machen. Der festzulegende Grenzwert ist sehr stark von dem zugrundeliegenden Dokument abhängig und kann nur durch intensive Tests bestimmt werden. Aufgrund der sehr effizienten Indexzugriffe wird er sich vermutlich in einem Bereich unter 15 Prozent einpendeln.

1 Insofern alle Texte eines Elements eine ähnliche Länge haben, d.h., die durchschnittliche Textlänge ist ein repräsentativer Wert.

2 (1 * 2 + 3 * 1 + 1 * 2) / (1 + 3 + 1) = 7 / 5 = 1,4

Allerdings ist in den Fällen, in denen indexintern Treffermengen zusammengeführt werden müssen, z.B. bei der Wildcardsuche, allgemein keine effiziente Bestimmung der Anzahl an Indextreffern möglich. Hierzu müssten die einzelnen Größen der indexinternen Treffermengen erst zusammengeführt werden. Die Anzahl der indexinternen Treffermengen hängt sehr stark von der Art der Wildcard ab. Hier werden vor allem ".+" und ".*" sehr viele, hingegen "." und ".?" nur wenige indexinterne Treffermengen erzeugen. Aber auch die Länge des Präfixes vor der Wildcard spielt hier eine Rolle. Baumbasierte Index-Strukturen können ihre Struktur mit steigender Länge des Präfixes immer besser ausnutzen und reduzieren so den Traversierungsaufwand bei der Wildcardsuche.

Somit können bei den Wildcards ".?" und "." analog zur einfachen Suche die Treffermengen vorab durch indexinterne Zusammenführung der Treffer ermittelt werden. Hingegen erscheint dieses Vorgehen für die Wildcards ".+" und ".*" nur effizient, wenn die Länge des Präfixes vor der Wildcard auf einen geringen Traversierungsaufwand schließen lässt. Einen genauen Grenzwert festzulegen erscheint jedoch schwierig und kann durch intensives Testen gefunden werden. Es ist auch denkbar, die Suffixlänge nach der Wildcard als weiteren Parameter zu verwenden.

Zusammenfassend lässt sich festhalten, dass der Aufwand für die Auswertung ohne Indexzugriff relativ gut abzuschätzen und für den Indexzugriff in den allermeisten Fällen möglich ist. Für die Entscheidung zwischen indexbasierter und sequentieller Auswertung ist das zuletzt gezeigte ausreichend, allerdings wird im nächsten Abschnitt bei der Zusammenfassung von Prädikaten der Path Summary eine zentrale Rolle zukommen.

6.2.3 Auswertungsstrategie für Pfadausdrücke mit mehreren Prädikaten

Bei der Auswertung von Pfadausdrücken mit mehreren Prädikaten ist zunächst das Prädikat zu evaluieren, das das kleinste Zwischenresultat ergibt. Dies ist von zentraler Bedeutung, da es als Basis für die Auswertung der folgenden Prädikate gilt. Dieses kann anhand des im letzten Abschnitt vorgestellten Verfahrens identifiziert werden. Die Pfadumkehrung ist, wie bei Anfragen mit nur einem Prädikat, analog anzuwenden. Allerdings kann nur bei Prädikaten, die keine Positionsfilter enthalten, die Reihenfolge beliebig verändert und damit u.U. das Prädikat mit der kleinsten Treffermenge nicht vorgezogen werden.

Die Verwendung der Index-Struktur findet, wenn überhaupt, nur beim ersten Prädikat Anwendung, sonst würde das Ergebnis des voranstehenden Prädikats nicht ausgenutzt, da sich der Index immer auf das gesamte Dokument bezieht. Die nun folgenden Prädikate werden sequentiell verarbeitet. In einige Fällen ist es darüber hinaus auch möglich Prädikate zusammenzufassen. Die Voraussetzungen und das Vorgehen wird im nächsten Abschnitt beschrieben.

6.3 Zusammenfassung von Prädikaten

Die hier getroffenen Überlegungen zur Zusammenfassung von Prädikaten beziehen sich rein auf Prädikate mit Full-Text Bezug, wie z.B.:

```
/descendant::MEDIUM[TIT ftcontains "david"][TIT ftcontains "gale"]
```

Hierbei handelt es sich um eine der häufigsten realen Anfragen aus den Log-Daten der Universitäts-Mediothek (siehe Abschnitt (3.1.1)). Die Anfrage wird durch die visuelle Suchmaske von MedioVis automatisch generiert. Der Anwender sucht hier offensichtlich ein Medium, dessen Titel, gespeichert in dem Element TIT, "david" und auch "gale" enthält.

Bei der Auswertung der Anfrage wird, wie in Abschnitt (6.2.3) beschrieben, zunächst das Prädikat mit der kleineren Treffermenge evaluiert, in diesem Fall das zweite Prädikat `[TIT ftcontains "gale"]` und auf das resultierende Zwischenergebnis das erste Prädikat `[TIT ftcontains "david"]` angewandt. Wird die Full-Text Index-Struktur zur Anfrageverarbeitung verwendet, werden alle Indextreffer geprüft, ob sie in einem TIT Element vorkommen, dann wird der Text des TIT-Elementes aus der Datenbank gelesen und sequentiell nach "david" durchsucht. Dieses Vorgehen erscheint auf den ersten Blick ineffizient, da man sowohl "gale" als auch "david" über die Index-Struktur suchen, die Treffer über ein `ftand` verknüpfen und hierauf die TIT Elementprüfung anwenden könnte. Dies entspräche der Anfrage:

```
/descendant::MEDIUM[TIT ftcontains "david" ftand "gale"]
```

Allerdings ist diese Anfrage nicht zwangsläufig äquivalent zur ursprünglich spezifizierten Anfrage, da laut Definition die Expression vor dem `ftcontains` eine Sequenz von Knoten liefert. Enthält diese Sequenz mehr als nur ein Item, d.h. ein Medium hat zwei oder mehr TIT-Elemente als Kinder, so müssen "david" und "gale" nicht zwangsläufig im gleichen TIT-Element vorkommen. Bei der zusammengeführten Anfrage müssen sie hingegen im gleichen TIT-Element vorkommen.

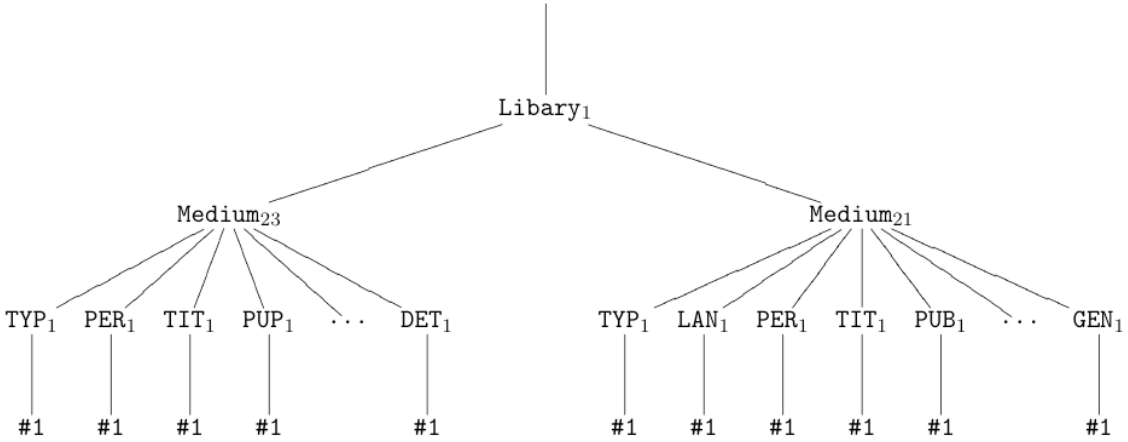
Anhand der Path Summary in Abbildung (6.5) ist zu erkennen, dass es kein Medium-Element gibt, das mehr als ein TIT-Element als Kind hat, und alle TIT-Elemente nur ein Textknoten als Kind haben. Daher sind in diesem speziellen Fall die Anfragen

`/descendant::MEDIUM[TIT ftcontains "david"][TIT ftcontains "gale"]` und
`/descendant::MEDIUM[TIT ftcontains "david" ftand "gale"]` äquivalent.

Das Zusammenfassen von Prädikaten ist auch möglich, wenn FTContains Expressions innerhalb eines Prädikates mit `and` oder `or` verknüpft werden. Das `or` kann dann in ein `ftor` überführt werden, die mehrfache Pfadauswertung in dem Prädikat wird eingespart.

Es lässt sich festhalten, dass Prädikate zusammengefasst werden können, wenn zum einen der gleiche Pfad in allen Prädikaten spezifiziert ist und zum anderen über Metainformationen, wie z.B. einer Path Summary, sichergestellt ist, dass durch das Zusammenfassen eine äquivalente Anfrage entsteht.

Abbildung 6.5: Path Summary (verkürzt) der Mediothek: t_n, t = Tag, n = Häufigkeit, #k = Anzahl Textknoten



KAPITEL 7

Visualisierung von XML-Full-Text Daten

7.1 Visualisierung von XML-Daten

In den voranstehenden Kapiteln ist die Speicherung und Verarbeitung von XML-Full-Text Daten detailliert dargestellt worden, genauso wie die Evaluierung von Anfragen auf diesen Daten. In diesem Kapitel wird hieran anknüpfend eine Visualisierung von Treffern solcher Anfragen vorgestellt. Zunächst wird jedoch eine allgemeine Visualisierungstechnik zur Darstellung von XML-Daten eingeführt, die in BaseX Anwendung findet, und in die nachstehend die Full-Text Visualisierung integriert wird.

Grundsätzlich können zur Darstellung von XML-Daten verschiedene Verfahren, die zur Visualisierung von hierarchischen Daten dienen, genutzt werden. Sehr häufig sind Anwendungen zu finden, die durch Einrücken der Zeilen des Dokuments versuchen, die Hierarchieebenen der Elemente darzustellen. Der textuelle Inhalt eines Dokuments bleibt so vollständig erhalten, lediglich seine Formatierung wird angepasst. Aber auch weiterführende Verfahren, die in der Regel den Inhalt des Dokuments nicht mehr vollständig darstellen, sind weit verbreitet.

Ein klassischer, aus der Literatur bekannter Ansatz zur Darstellung hierarchischer Daten sind baumbasierte Visualisierungen, die direkt die hierarchische Struktur durch einen Baum, bei dem immer zwei Knoten durch genau eine Kante verbunden sind, darstellen [24]. Die 2-dimensionalen Varianten der baumbasierten Visualisierungen haben jedoch den Nachteil, dass bereits bei wenigen Ebenen und einigen Knoten z.B. auf der dritten Ebene ausreichend viel Platz zur Darstellung zur Verfügung stehen muss. Daher versuchen die 3-dimensionalen Varianten, wie z.B. der Cone Tree [23], die zur Verfügung stehende Darstellungsfläche effizienter zu nutzen und mittels Interaktionselementen den Informationsverlust durch Überlagerung zu reduzieren. Hierbei werden die nachfolgenden Knoten eines gegebenen Knotens kreisförmig angeordnet, so dass sie zusammen mit dem übergeordneten Knoten einen Kegel bilden.

Beide Verfahren nutzen die zur Verfügung stehende Darstellungsfläche nicht vollständig aus und sind für den Anwender bei sehr großen Hierarchien nur schwer zu interpretieren. Baumbasierte Verfahren zielen primär auf die Darstellung der gesamten Hierarchiestruktur

ab und nur bei ausreichend zur Verfügung stehender Darstellungsfläche können weitere Informationen in textueller Form dargestellt werden.

Aufgrund dieser Einschränkungen hat Shneiderman die raumfüllende TreeMap [17] Visualisierung eingeführt. Sie ermöglicht die Darstellung komplexer hierarchischer Daten in einer dem Betrachter zugänglichen Weise. Zunächst wird für den Wurzelknoten ein Rechteck gezeichnet und innerhalb dieses Rechtecks wird für jeden Kindknoten ein eigenes Rechteck eingefügt. Dieser Prozess setzt sich fort, bis alle Knoten der hierarchischen Struktur abgearbeitet sind. Auf die Hintergrundfarbe der Rechtecke kann ein weiteres Attribut gemapt werden [24]. Die X- und Y-Dimensionen der Darstellungsfläche werden alternierend aufgrund der Attributwerte partitioniert und bestimmen so die Größe der Rechtecke. Die Auswahl und Reihenfolge der Attributwerte, aufgrund derer partitioniert wird, kann beliebig gewählt werden. Allerdings können hierdurch sehr ungünstige Seitenverhältnisse der Rechtecke entstehen oder es kann auch ihre Ordnung verloren gehen. Daher sind eine Reihe von Erweiterungen dieses Ansatzes bekannt, die versuchen, unterschiedlichste Nebenbedingungen bei der Konzeption der TreeMap mit zu berücksichtigen. Diese werden in der vorliegenden Arbeit jedoch nicht weiter betrachtet.

Der TreeMap Algorithmus arbeitet interessanterweise sehr ähnlich, wie die Algorithmen zur Traversierung von XPath Achsen [13]. Ein sequentieller Scan der Datenbanktabelle[1] erlaubt die direkte Berechnung der zu zeichnenden Rechtecke. Durch das sequentielle Hinzufügen der Rechtecke werden diese nach ihrem pre-Wert sortiert angeordnet. Daher folgen nach einem Knoten alle im Dokumentenbaum nachfolgenden Knoten (descendants). In Abbildung (7.1) ist die TreeMap Visualisierung der 0.1 MB Wikipedia Instanz dargestellt.

Jedes Element wird als Rechteck dargestellt und der Tag des Elements als Text im oberen linken Bereich des Rechtecks angezeigt. Es ist sehr gut zu erkennen, dass am Anfang der Datei ein `siteinfo` Element mit Metainformationen zu finden ist, dem dann die einzelnen `page` Elemente folgen, die jeweils einem Wikipedia Artikel entsprechen. Auch die Struktur der `page` Elemente ist sehr gut zu erkennen. Diese enthalten jeweils ein `title`, `id` und `revision` Element, welches wiederum eine Reihe weiterer Elemente enthält - beispielsweise das `text` Element, welches den Full-Text des Artikels fasst. Die TreeMap ermöglicht es, die gesamte Struktur des Dokuments zu erfassen und nutzt dabei die zur Verfügung stehende Darstellungsfläche voll aus. Durch Interaktion, z.B. zoomen, kann ein Teilbereich der TreeMap ausgewählt und mit einem höheren Detailgrad dargestellt werden, so dass eine explorative Analyse des Dokuments möglich ist. Die mit den Knoten als Treffer von Anfragen in Beziehung stehenden Rechtecke werden durch eine entsprechende Markierung farblich abgehoben.

Allerdings wird in Abbildung (7.1) auch sehr gut deutlich, dass der zur Verfügung stehende Platz in den Rechtecken auf der untersten Ebene stark eingeschränkt ist und sich daher die Full-Texte der Elemente nicht immer direkt darstellen lassen. Bei einem der `namespace` Elemente ist der Full-Text "user" beispielhaft zu erkennen. Um diesen jedoch

1 Beispielhaft dargestellt in Abschnitt (5.3.1).

Abbildung 7.1: TreeMap Visualisierung der 0.1 MB Wikipedia Instanz

bei allen Elementen darstellen zu können und vor allem auch Treffer in den enthaltenen Full-Texten zu visualisieren, sind weiterführende Konzepte als die einfache Darstellung der Full-Text erforderlich. Ferner ist es wünschenswert, dem Betrachter Zusatzinformationen, die über die einfache Markierung des umgebenden Rechtecks hinaus gehen, zur Verfügung zu stellen.

7.2 Visualisierung von XML-Full-Text Daten

7.2.1 Gewinnung von XML-Full-Text Daten

Nun stellt sich die Frage, wie die zu visualisierenden Daten gesammelt und an die GUI weitergereicht werden können. Die Full-Text Operatoren liefern für jeden Trefferknoten dessen ID (pre-Wert) und die Tokenpositionen (pos-Werte) der Suchbegriffe aus der

Anfrage in dem Knotentext.[1] Zusätzlich wird zu jedem pos-Wert eine Referenz gespeichert, die anzeigt, zu welchem Token der Anfrage der jeweilige pos-Wert gehört. Die weitere Verarbeitung dieser Daten hängt nun von dem gewählten Verarbeitungsmodus ab.

Sequentieller Verarbeitungsmodus - ohne Pfadumkehrung

Im sequentiellen Verarbeitungsmodus (siehe Abschnitt (4.2)) wie auch im sequentiell indexbasierten Verarbeitungsmodus (siehe Abschnitt (4.3.2)), die beide keine Pfadumkehrung (siehe Abschnitt (4.3.1)) vollziehen, resultieren aus der Full-Text Verarbeitung nur Treffer, die auch dem Pfadausdruck entsprechen, da dieser zuvor geprüft wurde. Somit ist sichergestellt, dass die Full-Text Verarbeitung nur Textknoten, die auch dem Pfadausdruck vor dem Prädikat entsprechen, evaluiert. Allerdings können nach dem Prädikat, das den Full-Text Ausdruck enthält, beliebig viele weitere Steps folgen und deren jeweilige Evaluation sich der des Full-Text Ausdrucks anschließt.

```
/a[b/text() ftcontains "z"]/d
```

In dem oben aufgeführten Beispiel wäre das der Step `/d`, da dieser dem Full-Text Prädikat `[b/text() ftcontains "z"]` folgt. Hier müsste jetzt der Step `/d` die Full-Text Daten, die das Prädikat geliefert hat, verwalten und bereinigen, so dass nur noch Full-Text Daten als Treffer zurückgegeben werden, die auch dem `/d` Step entsprechen. Da einem Prädikat ein beliebiger Pfadausdruck folgen kann, sind alle an dieser Stelle aufgrund der Grammatik zugelassenen Steps für die Verarbeitung von Full-Text Daten zu erweitern.[2] Im Rahmen dieser Arbeit wird aufgrund der Komplexität hierauf nicht weiter eingegangen. Daher werden bei der Visualisierung zwar alle Full-Text Treffer solcher Anfragen visualisiert, jedoch nur die Knoten, die der Anfrage auch tatsächlich entsprechen, gesondert visuell hervorgehoben.

Indexbasierter Verarbeitungsmodus - mit Pfadumkehrung

Findet die in Abschnitt (4.3.1) beschriebene Pfadumkehrung Anwendung, so wird das Full-Text Prädikat zunächst ausgewertet und anschließend der invertierte Pfad. Aufgrund dieser Vorgehensweise treffen die hier beschriebenen tiefgreifenden Auswirkungen auf die Evaluation der Steps, deren Berechnung dem Full-Text Prädikat folgt, ebenso zu.

And und Or Expressions

Sind innerhalb eines Full-Text Prädikates `And` oder `Or` Expressions verwendet worden, so müssen auch die Full-Text Daten weiter verarbeitet werden. Bei einer `And` Expression sind die Full-Text Daten der Operanden über ihre pre-Werte zu vereinigen, d.h., kommt ein pre-Wert in den Full-Text Daten eines jeden Operanden vor, so ist dieser auch zu visualisieren. Bei der `Or` Expression sind die Full-Text Daten der Operanden zu sammeln und anschließend zu visualisieren. Sollten in der Anfrage mehrere Prädikate hintereinander spezifiziert sein, so ist analog zu der `And` Expression vorzugehen.

1 Details hierzu sind Abschnitt (5.2) zu entnehmen.
2 Dies ist analog zu der Handhabung von Scoring-Werten in Pfadausdrücken zu sehen.

7.2.2 Grenzen klassischer Text Visualisierung im TreeMap Kontext

Bei der in Abschnitt (7.2.1) vorgestellten TreeMap Visualisierung hat sich bereits gezeigt, dass die zur Verfügung stehende Darstellungsfläche in den Rechtecken der TreeMap stark eingeschränkt ist. Um in den Rechtecken Texte in ihrer ursprünglichen Form darstellen zu können, bietet sich die Möglichkeit durch Interaktion, z.B. Zoomen, die Darstellungsfläche zu vergrößern. Allerdings kann nicht sicher gestellt werden, dass durch Zoomen die Darstellungsfläche ausreichend groß ist, um den gesamten Text darzustellen. Häufig kann nur ein Ausschnitt des Textes dargestellt werden, so dass nur durch Scrolling die nicht darstellbaren Teile des Textes erreichbar sind. Durch Zoomen geht jedoch der Kontextbezug des gezoomten Rechtecks verloren. Bei der Suche nach Begriffen in dem Text ist es nicht möglich, einen Überblick über die Verteilung aller Treffer in dem gesamten Text zu erhalten. Vielmehr ist dieser begrenzt auf den darstellbaren Textausschnitt und erst durch Interaktion kann der Betrachter nach weiteren Treffern in dem Text suchen.

Es ist denkbar, durch die Anpassung der Schriftgröße die Menge an darstellbarem Text auf der begrenzt zur Verfügung stehenden Darstellungsfläche zu erhöhen. Jedoch kann die Schriftgröße nicht beliebig verkleinert werden, da der Text ab einer gewissen Größe nicht mehr für den Betrachter wahrnehmbar ist. Daher ist eine Visualisierung des Textes notwendig, die die vorhandene Darstellungsfläche optimal nutzt, den gesamten Text visualisiert und soweit möglich dessen Struktur erhält. Im Folgenden wird sich zeigen, dass das dort eingeführte Visualisierungverfahren diese Kriterien erfüllt.

7.2.3 Thumbnail Visualisierung für Full-Text

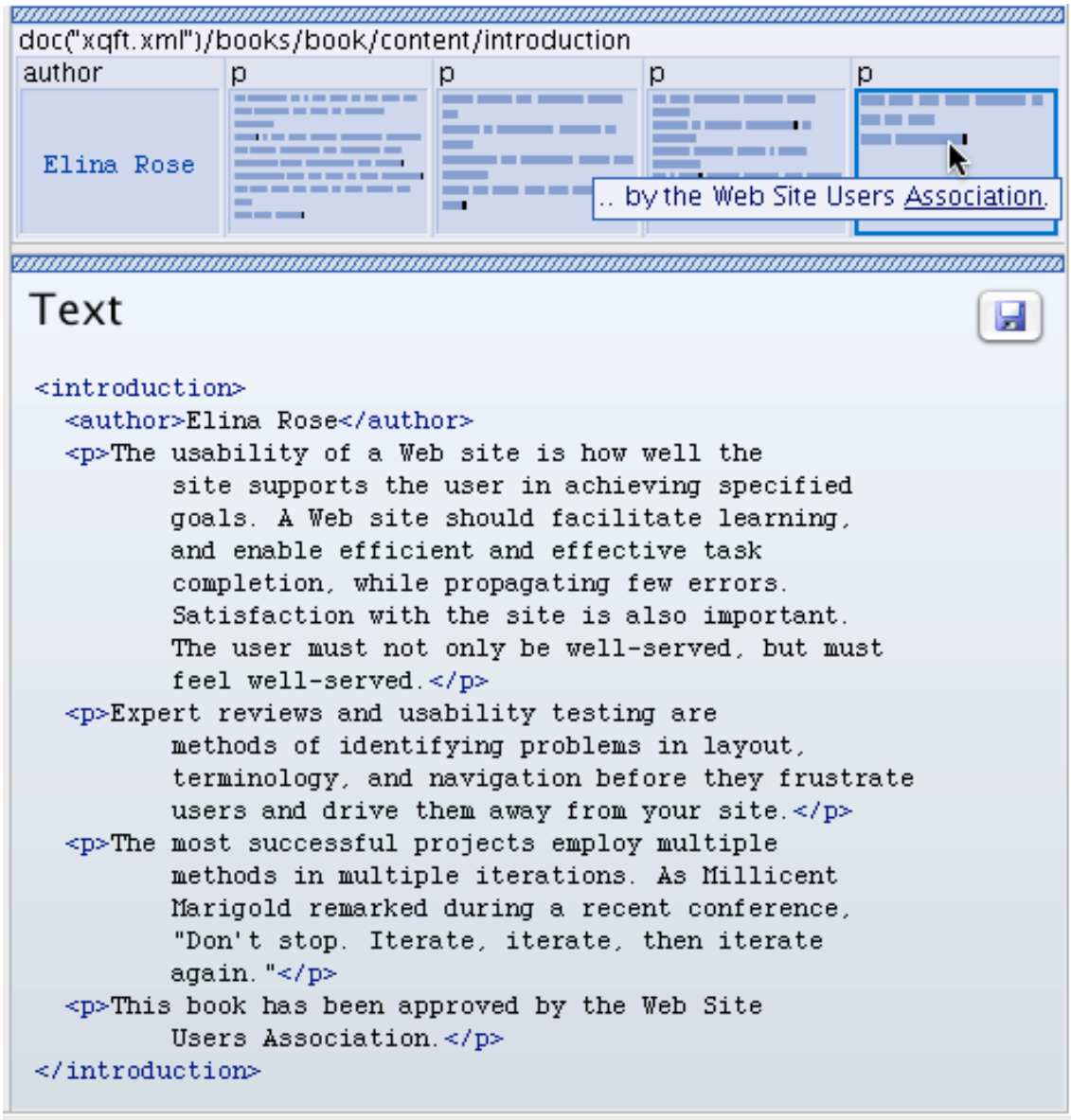
Adaption der Klassischen Thumbnail Visualisierung

Eine Visualisierung, bei der anstelle des reinen Textes grafische Repräsentatoren (Thumbnails) für die einzelnen Token des Textes verwendet werden, hat Kaugars (1989) vorgestellt [18]. Der Text wird sequentiell verarbeitet und anstelle jedes Tokens ein entsprechender Thumbnail dargestellt. Die Größe des Thumbnails richtet sich nach der Größe des repräsentierten Texttokens [21]. Zeilenumbrüche zwischen den Tokens des Textes bleiben in der Thumbnail Visualisierung erhalten.

Um die Struktur des Textes zu erhalten, kann die von Kaugars eingeführte Thumbnail Visualisierung erweitert werden, indem Satzzeichen wie auch Leerzeichen zwischen den Token des Textes visualisiert werden. Allerdings macht die Visualisierung von Zeilenumbrüchen im XML-Full-Text Kontext wenig Sinn, da hier in der Regel Zeilenumbrüche aus technischen Gründen in den Full-Texten zu finden sind und nicht der Textformatierung dienen. In Abbildung (7.2) ist im oberen Bereich die resultierende TreeMap mit der Thumbnail Visualisierung der Full-Texte aus dem darunter stehenden Dokumentenfragment dargestellt. Es ist sehr gut zu erkennen, wie die manuell eingefügten Zeilenumbrüche die Ausrichtung der Thumbnails beeinflussen und die flüssige Repräsentation des Textes aufgrund deren Berücksichtigung gestört ist.

Der Text des `autor` Elements passt sehr gut in das entsprechende Rechteck, daher ist er als einfacher Text dargestellt. Bei den `p` Elementen passt der Full-Text hingegen nicht

Abbildung 7.2: TreeMap mit Thumbnail Visualisierung der Full-Texte und Tooltip



in die Rechtecke, daher wird hier die Thumbnail Visualisierung eingesetzt. Die schwarzen Thumbnails stehen hier als Repräsentanten für Satzzeichen, die einen Satz beenden. Da die Länge eines Thumbnails die Tokenlänge des repräsentierten Tokens abbildet, bleibt die Struktur des Dokuments sehr gut erhalten. Durch den Einsatz der Thumbnails geht der ursprüngliche Text an dieser Stelle verloren. Um daher den Bezug hierzu wieder aufzubauen, ist eine Tooltip Funktionalität ergänzt worden - die genauer in Abschnitt (7.2.5) beschrieben wird.

Die Abbildung (7.2) zeigt beispielhaft im oberen rechten Bereich die Tooltip Funktiona-

lität. Hier wird das Token, dessen Thumbnail-Repräsentant aktuell durch den Mauszeiger ausgewählt ist, in dem Tooltip-Text unterstrichen dargestellt. In der Abbildung (7.2) handelt es sich um das Token "Association". Zusätzlich werden noch einige Tokens vor und nach dem angewählten Token dargestellt, um den Kontext aufzugreifen. So kann durch sequentielles Überlaufen der Thumbnails der gesamte Text reproduziert und gelesen werden. Allerdings ist es offensichtlich, dass bei einem etwas längerem Text - trotz der Thumbnail Visualisierung - die zur Verfügung stehende Darstellungsfläche nicht ausreicht. Daher ist ein dynamisches Verfahren notwendig, das die Parameter der Thumbnail Visualisierung (Größe der Thumbnails und deren Abstand) so anpasst, dass die bestehende Darstellungsfläche optimal genutzt wird.

Dynamische Anpassung der Thumbnail Parameter

Um die Thumbnail Parameter Länge, Breite, Höhe, vertikaler und horizontaler Abstand für einen gegebenen Text, der in einem Treemap Rechteck mittels Thumbnail Visualisierung dargestellt werden soll, bestimmen zu können, lassen sich zwei grundsätzlich unterschiedliche Ansätze unterscheiden. Zunächst einmal kann versucht werden, den Platzbedarf über eine Schätzung zu bestimmen. Die gegebene Textlänge multipliziert mit der Breite eines Thumbnails errechnet die Gesamtlänge der einzelnen Thumbnails. Teilt man diese durch die Breite des Rechtecks, so erhält man die Anzahl an Zeilen, die notwendig sind. Unter Berücksichtigung eines Zeilenabstandes und der Höhe der Thumbnails errechnet sich so der Platzverbrauch für einen gesamten Text. Allerdings ist diese Schätzung äußert ungenau, da sie davon ausgeht, dass Begriffe bzw. deren Thumbnailrepräsentanten am Ende jeder Zeile getrennt und in der nächsten Zeile fortgesetzt und so nicht komplett umgebrochen werden. Hinzu kommt, dass im Text vorhandene Absätze nicht berücksichtigt werden. Nun wäre es denkbar, durch eine zugelassene Toleranz den Platzverbrauch noch ungenauer zu schätzen. Aber selbst diese Schätzung kann nicht garantieren, dass der geschätzte Platzverbrauch den realen nicht unterschätzt und so mit den gefundenen Parametern die Darstellungsfläche nicht ausreicht. Daher ist eine genaue Bestimmung des Platzverbrauchs unumgänglich.

Der exakte Platzverbrauch lässt sich am einfachsten bestimmen, indem die Thumbnail Visualisierung mit einer gegebenen Parametereinstellung für den zu visualisierenden Text berechnet wird und so deren Platzverbrauch erhält. Nun müssen die Parameter so angepasst werden, bis der errechnete Platzverbrauch geringer als die vorhande Darstellungsfläche ist und diese möglichst voll ausnutzt.

Ausgehend von einer Startparametereinstellung kann durch iteratives Anpassen der Parameter eine Parametereinstellung gefunden werden, die garantiert, dass die Thumbnails Visualisierung sich auf der gegeben Darstellungsfläche darstellen lässt. Allerdings hängt die Ausnutzung der zur Verfügung stehenden Darstellungsfläche sehr stark von dem Faktor ab, mit dem die Parametereinstellungen iterativ angepasst werden. Hier muss immer ein Kompromiss zwischen Anzahl der Iterationen und optimaler Flächenausnutzung gefunden werden.

Alternativ empfiehlt es sich, einen binären Ansatz zu verfolgen. Hierbei wird der Platz-

verbrauch zu einer Startparametereinstellung errechnet, übersteigt dieser die zur Verfügung stehende Darstellungsfläche, so werden die Parameter halbiert und der Platzverbrauch erneut errechnet. Ist dieser nun kleiner als die zur Verfügung stehende Fläche, kann der Platzverbrauch erneut mit den Parametern als Mittelwerte zwischen alten und aktuellen Werten errechnet werden. Übersteigt der neu errechnete Platzverbrauch die Darstellungsfläche, so ist die letzte Parametereinstellung zu wählen. Diese Vorgehensweise ist analog zur binären Suche zu sehen und garantiert, in logarithmischer Zeit eine passende Parametereinstellung zu finden [9].

Abbildung 7.3: Binäre Thumbnailparameter Anpassung - Übersicht über mögliche Rechtecksgrößen

Abbildung 7.4: Binäre Thumbnailparameter Anpassung - gesuchte Thumbnailparameter zum Rechteck mit "?"

Die Abbildungen (7.3), (7.4), (7.5) und (7.6) veranschaulichen das Vorgehen noch einmal. In der Abbildung (7.3) sind die resultierenden Thumbnail Visualisierungen zu unterschiedlichen Parametereinstellungeng zu sehen. Ist nun eine Rechtecksgröße vorgegeben, beispielhaft die des Rechtecks mit dem "?" in Abbildung (7.4), so wird mit der Startparametereinstellung die resultierende Größe des Rechtecks berechnet.

Abbildung 7.5: Binäre Thumbnailparameter Anpassung - erster Schritt

Da die gewünschte Zielgröße kleiner ist als die resultierende, werden die Parametereinstellungen halbiert und das Rechteck neu berechnet. Das entstehende Rechteck ist allerdings immer größer als die zur Verfügung stehende Darstellungsfläche (Abbildung (7.5)), so dass die Parameter erneut anzupassen sind und wiederum eine Berechnung der

Größe des Rechtecks stattfindet.

Abbildung 7.6: Binäre Thumbnailparameter Anpassung - zweiter Schritt

Das sich nun ergebende Rechteck hat die Größe des gesuchten Rechtecks (Abbildung (7.6)) und so ist die Parametereinstellung gefunden, mit der sich dieses Rechteck optimal füllen lässt. Allerdings wird auch klar, dass bei genügend langen Texten und nur wenig zur Verfügung stehender Darstellungsfläche die Thumbnails so klein werden, dass sie für den Betrachter kaum noch also solche wahrzunehmen sind. Daher sind Grenzwerte für die minimale Größe festzulegen. Allerdings ergibt sich hieraus eine Längenbeschränkung für die darzustellenden Texte. Um diese zu umgehen, kann eine weitere Abstraktionsebene für die Thumbnails eingeführt werden.

Thumbnails als Repräsentanten für Sätze

Im Voranstehenden wurden Thumbnails als Repräsentanten für einzelne Tokens eingeführt und auf die Längenbeschränkung aufgrund der Mindestgröße einzelner Thumbnails hingewiesen. Repräsentieren Thumbnails nun einen kompletten Satz, so entfällt diese Beschränkung weitgehend. Die Länge eines Thumbnails richtet sich nach der Summe aller Tokenlängen, die in dem Satz enthalten sind. Bei der Visualisierung wird der notwendige Platz für die Darstellung der Leerzeichen zwischen den einzelnen Tokens eingespart. In Abbildung (7.7) ist in den beiden linken Rechtecken die tokenbasierte Thumbnail Visualisierung zu sehen, im mittleren Rechteck wurden die Thumbnails aufgrund der Satzlänge berechnet, wobei jedoch in diesem Fall die Zeilenumbrüche erhalten bleiben.

Abbildung 7.7: Thumbnail als Repräsentatoren für Tokens beziehungsweise Sätze

Durch den Erhalt der Zeilenumbrüche bleibt die ursprüngliche Struktur des Textes sehr gut zu erkennen. Allerdings wird hierdurch die zur Verfügung stehende Darstellungsfläche nicht vollständig ausgenutzt. Daher bietet es sich an, die Zeilenumbrüche aus dem Text zu entfernen und so die Darstellungsfläche vollständig zu nutzten. Die resultierende Thumb-

nail Visualisierung ist den beiden rechten Rechtecken in Abbildung (7.7) zu entnehmen. Aufgrund der Visualisierung der Satzzeichen bleibt die Satzstruktur des Textes erhalten. Die notwendige Darstellungsfläche der ursprünglich tokenbasierten Darstellung hin zur satzbasierten hat sich auf ca. ein Drittel reduziert.

In einem weiteren Schritt ist es nun denkbar, anstelle von Sätzen ganze Absätze durch einen Thumbnail zu repräsentieren. Hierbei würde dann jegliche Satzstrukturinformation innerhalb eines Absatzes verloren gehen. Der Informationsgehalt der Visualisierung bietet dann nur noch einen Längenvergleich der einzelnen Absätze. In Abbildung (7.8) ist der Unterschied sehr deutlich zu erkennen.

Abbildung 7.8: Satz- und Absatzbasierte Thumbnail Visualisierung

Das obere Rechteck vermittelt nur einen Eindruck über die Länge des Absatzes - in diesem Fall enthält der Knoten nur einen Absatz - im unteren Rechteck sind die unterschiedlichen Satzlängen innerhalb des Absatzes sehr gut zu erkennen. Eine absatzbasierte Thumbnail Repräsentation macht damit nur Sinn, wenn der Text mehrere Absätze enthält - bei den Wikipedia Instanzen bestehen alle Full-Texte aus nur einem Absatz. Dies ist für Full-Texte in XML-Dokumenten typisch, da die Formatierung des Textes durch Absätze, um z.B. deren Lesbarkeit zu erhöhen, nur selten genutzt wird. Vielmehr wird die variable Struktur von XML-Dokumente genutzt, um jeden einzelnen Absatz auch in ein getrenntes Element zu schachteln und dieses unter Umständen durch Zusatzinformationen, z.B. in Form von Attributen, zu ergänzen. Aus diesem Grund werden in dieser Arbeit keine Thumbnails als Repräsentanten für Absätze verwendet - allerdings kann dies in einem anderen Kontext durchaus sehr sinnvoll sein. Im Folgenden wird gezeigt, wie für jedes Rechteck die Abstraktionsebene und Parameterkonfiguration gefunden werden kann, so dass die zur Verfügung stehende Darstellungsfläche optimal ausgenutzt wird.

Dynamische Auswahl der Abstraktionsebene

Zunächst einmal kann für das Rechteck unter Verwendung des in "Dynamische Anpassung der Thumbnail Parameter" eingeführten Verfahrens festgestellt werden, ob mit dem Abstraktionslevel "Token" die zur Verfügung stehende Darstellungsfläche ausreichend ist. Ist dies nicht der Fall, ist das Abstraktionslevel auf "Satz" umzustellen und das Verfahren beginnt mit den zurückgesetzten Parameter von neuem. Da hier zunächst die Zeilenumbrüche erhalten bleiben sollen und keine Tokentrennung am Zeilenende stattfindet, kann der Platzverbrauch nicht direkt exakt errechnet werden. Sollte auch die Wiederholung des Verfahrens der dynamischen Anpassung der Thumbnail Parameter keine passende Lösung liefern, da die Darstellungsfläche für die gefunde Visualisierung zu klein ist, so sind die Parameter wie folgt manuell zu berechnen.

Der Zeilenabstand zwischen den Thumbnails ($lineHeigth$) und die damit verbundene Höhe wird auf einen von der festgelegten Schriftgröße für die Darstellung von Texten abhängigen Minimalwert gesetzt. Nun kann die Thumbnailbreite ($thumbWidth$) für ein Zeichen berechnet werden:

```
1: tLength ← Länge des darzustellenden Textes
2: numPM ← Anzahl Satzzeichen im darzustellenden Text
3: nl ← (int) (rectHeight / lineHeigth )
4: thumbWidth ← ( nl * rectWidth - numPM) / tLength
```

Mit der berechneten $thumbWidth$ kann nun die Thumbnail Visualisierung für das gegeben TreeMap Rechteck berechnet werden und es ist sichergestellt, dass die zur Verfügung stehende Darstellungsfläche in dem TreeMap Rechteck ausreichend ist. Da Zeilenumbrüche keine Berücksichtigung mehr finden und Tokens am Zeilenende getrennt und in der nächsten Zeile fortgesetzt werden - ohne Rücksicht auf korrekte Silbentrennung - kann so der Platzverbrauch exakt berechnet werden.

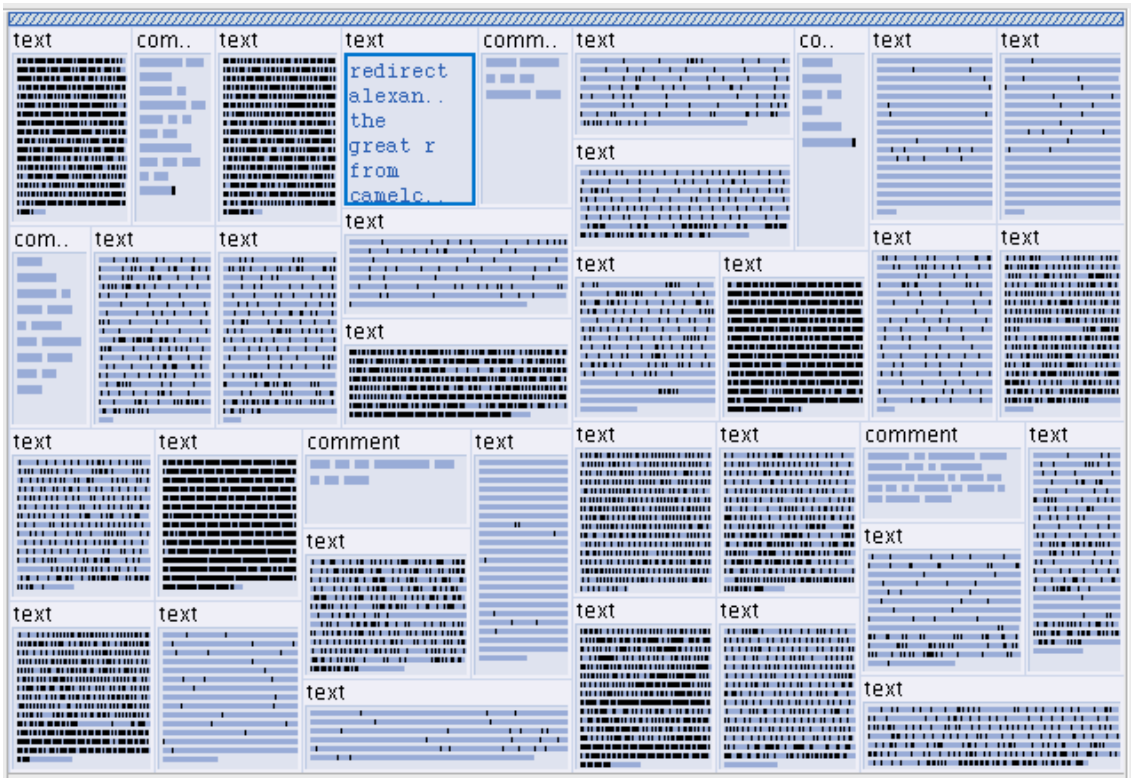
Visualisierung von Full-Text-Daten mit dynamischer Abstraktionsebene

In Abbildung (7.9) ist ein Ausschnitt von 35 Elementen der 1 MB Wikipedia Instanz zu sehen, der sehr übersichtlich die unterschiedlichen Abstraktionsebenen des eingeführten Visualisierungsverfahrens zeigt. In einem der TreeMap Rechtecke ist der Platz ausreichend, um den gesamten Text "redirect alexander the great r from camelcase" des Elements darzustellen. Da das Rechteck zu schmal ist, um die Tokens "alexander" und "camelcase" vollständig zu fassen, werden diese jeweils durch "alexan.."und "camelc.." verkürzt angezeigt. Die Texte der `comment` Elemente können als reiner Text nicht dargestellt werden - hier findet die tokenbasierte Thumbnail Visualisierung Anwendung. Die Struktur der Texte bleibt durch die Proportionalität von Token- zur visualisierten Thumbnaillänge sehr gut erhalten. Es lässt sich auch sehr gut erkennen, dass einer der Texte mit einem Satzzeichen endet (schwarzes Rechteck).

Bei den Rechtecken, die eine satzbasierte Thumbnail Visualisierung enthalten, ist bei mittellangen Texten die Verteilung der Satzlängen noch gut wahrnehmbar. Je dichter die schwarzen Rechtecke innerhalb eines TreeMap Rechtecks zusammenliegen, desto dunkler wird der visuelle Eindruck des gesamten Rechtecks. Dies deutet auf sehr lange Texte hin, bei denen die dargestellte Satzlänge (Breite der Thumbnail) soweit angepasst worden ist, dass deren Satzzeichen sehr nahe nebeneinander liegen. Trotz dieser Komprimierung bleibt die Struktur des Textes erhalten, und so lassen sich noch Aussagen über die Verteilung der Satzlängen in dem Text treffen. So ist z.B. sehr gut zu erkennen, dass jeweils der letzte Satz der Full-Texte der `text` Elemente mit keinem Satzzeichen endet. Die Ursache hierfür liegt an den am Ende der Wikipedia Artikel angehängten Metainformationen über die Kategorie des jeweiligen Textes.

Im Nachstehenden wird weiterführend beschrieben, wie sich Treffer von Suchanfragen in der eingeführten Visualisierung entsprechend hervorheben lassen, so dass sie von dem Betrachter einfach und schnell zu erfassen sind.

Abbildung 7.9: TreeMap mit Text/Thumbnail Visualisierung



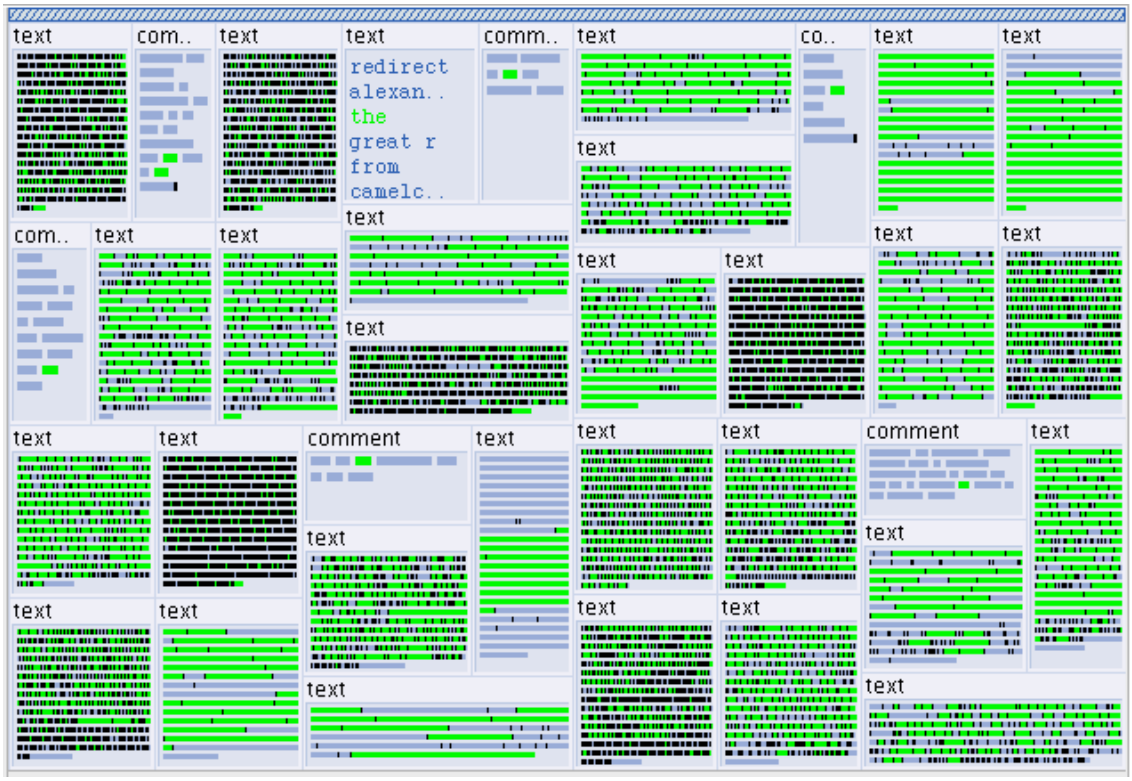
7.2.4 Visualisierung von Full-Text Suchergebnissen

Einfache Suche - FTWord

Bei der einfachen Suche nach nur einem Suchbegriff, dessen Resultat beispielhaft in Abbildung (7.10) dargestellt ist, liegt die Anfrage `//*[text() ftcontains "the"]` zugrunde, bei der alle Textknoten der 1 MB Wikipedia Instanz durchsucht werden. Die Visualisierung der Treffer in den einzelnen TreeMap Rechtecken hängt von dem in dem Rechteck gewählten Abstraktionslevel für die Textvisualisierung ab. Wird der Text in seiner ursprünglichen Form, also für den Anwender direkt lesbar, dargestellt, so kann auch der Suchbegriff direkt durch Einfärbung (`the`) hervorgehoben werden (erste Zeile, viertes Rechteck). Analog kann der Thumbnail Repräsentant eingefärbt werden, wenn Tokens als Abstraktionslevel gewählt wurden (erste Zeile, zweites Rechteck).

Bei der Repräsentation ganzer Sätze durch einen Thumbnail wird hingegen der gesamte Satz, in dem der Suchbegriff vorkommt, eingefärbt. Alternativ wäre denkbar, nur den Teil der Thumbnail einzufärben, die den Suchbegriff in dem Satz repräsentiert. Allerdings würde dies bei langen Texten dazu führen, dass durch die Komprimierung der Satzlänge der eingefärbte Teil der Thumbnail nicht mehr wahrzunehmen ist (erste Zeile, erstes Rechteck). Die Visualisierung in Abbildung (7.10) veranschaulicht die Vorteile des gewählten Konzepts sehr deutlich, so dass auch bei langen Texten die Sätze, die Treffer enthalten, noch sehr gut

Abbildung 7.10: TreeMap mit Text/Thumbnail Visualisierung - Suchergebnisse von `the`



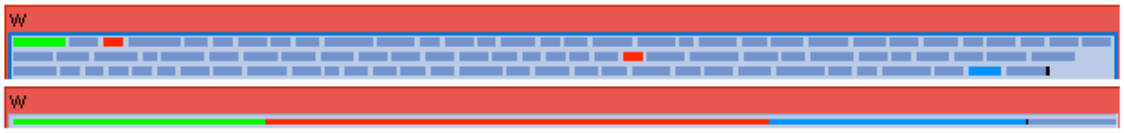
wahrzunehmen sind. Der Verlust an Detailinformation über den Aufbau jedes einzelnen Satzes ist an dieser Stelle zu vernachlässigen, da bei langen Texten und begrenzter Darstellungsfläche der Focus auf der Satzstruktur des gesamten Textes liegt.

FTAnd und FTOr

Bei der Visualisierung von Treffern aufgrund einer Suche nach mehreren Suchbegriffen ist grundsätzlich analog zur einfachen Suche vorzugehen. D.h., jeder Thumbnail der Tokens, die als Treffer evaluiert wurden, wird entsprechend eingefärbt. Bei der satzbasierten Thumbnail Visualisierung wird dementsprechend der gesamte Thumbnail eingefärbt, sollte der durch ihn repräsentierte Satz einen Treffer enthalten. Sofern mehrere Treffer in einem Satz auftreten, wird die Thumbnaillänge anteilig durch die Anzahl der Treffer geteilt und jedes der entstehenden Segmente eingefärbt. Alternativ wäre es denkbar, die Länge der Segmente aufgrund der zugrunde liegenden Tokenlänge zu gewichten. D.h., das Segment eines längeren Tokens fällt entsprechend größer aus, als dass eines kürzeren Tokens. Allerdings lässt sich so keine Aussage mehr über die Häufigkeit des Vorkommens eines Tokens in einem Satz treffen und es kann zu sehr ungünstigen Verteilungen kommen, so dass kürzere Tokens kaum noch wahrzunehmen sind. Bekommt jedes Token unabhängig von seiner Länge den gleichen Anteil von der zur Verfügung stehenden Thumbnaillänge, so kann aufgrund der Farbverteilung eine Aussage über die Häufigkeit des Tokens in dem

Satz getroffen werden. Das nachfolgende Beispiel zeigt dies anschaulich.

Abbildung 7.11: TreeMap Rechteck mit Text/Thumbnail Visualisierung - vier Treffer in einem Satz



Das beschriebene Vorgehen ist in Abbildung (7.11) noch einmal dargestellt und zeigt im oberen Bereich die Verteilung der Begriffe `necessities`, `true` und `officer` in dem dargestellten Satz. Der untere Bereich zeigt den Wechsel des Abstraktionslevels von Token auf Satz, so dass der Thumbnail anteilig durch die Anzahl der Treffer geteilt wird. Da der Begriff `true` zweimal vorkommt, ist das rote Segment der Thumbnail doppelt so lang wie das grüne bzw. blaue Segment. Auch die Reihenfolge der Begriffe bleibt bei der Einfärbung der Segmente erhalten.

FTMildNot und FTUnaryNot

FTMildNot Ausdrücke bestehen immer aus einem Anfrageteil, der spezifiziert, welche Tokens in einem Text enthalten sein müssen, und einem Teil, der Tokens spezifiziert, die nicht enthalten sein dürfen. FTUnaryNot hingegen bestehen immer nur aus einer Negation. Dieser negierte Teil kann in der Visualisierung nicht berücksichtigt werden, da nur eine farbliche Hervorhebung möglich ist, wenn auch mindestens ein Token zum Hervorheben vorhanden ist. Daher werden bei einem FTMildNot Ausdruck auch nur Tokens farblich hervorgehoben, die in dem Anfrageteil vor dem `not in` auftreten. Gleiches gilt für Anfragen, die einen FTUnaryNot Ausdruck enthalten. Hier können auch nur die Tokens visuell hervorgehoben werden, die in dem Text auch real vorkommen und nicht in der Anfrage negiert spezifiziert sind, da in diesem Fall alle anderen Tokens, bis auf die negierten, einzufärben sind und damit kein Mehrwert an Informationen für den Betrachter entsteht.

7.2.5 Visualisierung von Full-Text Suchergebnissen mit Kontextbezug

Die in Abschnitt (7.2.3) eingeführte Visualisierung für Full-Texte kann effizient berechnet werden und bietet dem Anwender schnell einen Überblick über die Struktur des gesamten Textes beziehungsweise der einzelnen Sätze. Jedoch geht durch die Repräsentation der Tokens durch die Thumbnails die ursprünglich textuelle Information verloren. Um diesen Verlust zu kompensieren, kann der ursprüngliche Text durch Integration einer interaktiven Komponente mit der Visualisierung in Bezug gesetzt werden.

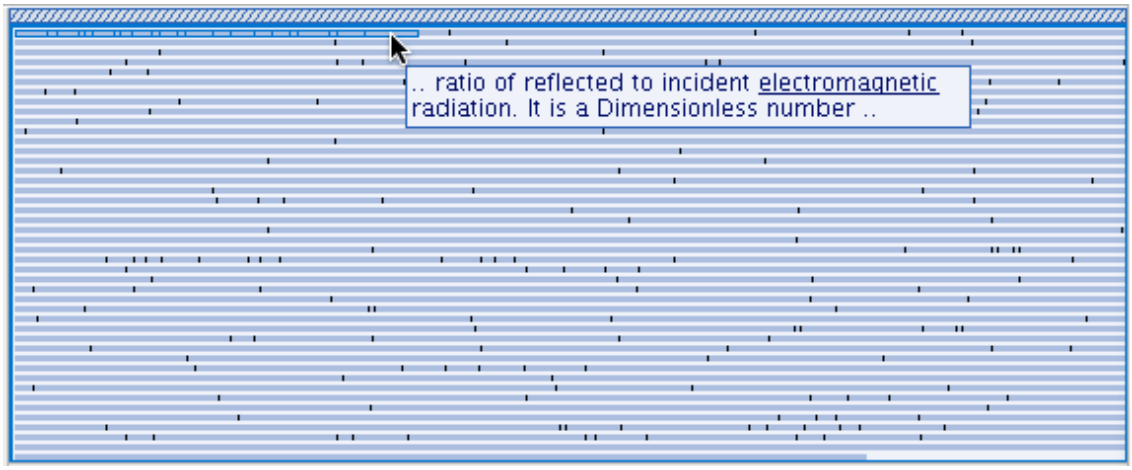
In diesem Zusammenhang hat sich das Konzept der Magic Lens [25], bei dem durch die Anwendung eines Operators die Sicht auf eine Region eines Objektes lokal verändert werden kann, bewährt. Dieser stellt auf Wunsch Detailinformationen zu Verfügung, ohne

den Kontext zu verlassen - im Gegensatz zu einem klassischen Zooming. Ein typisches Anwendungszenario für eine Magic Lens sind digitale Karten, bei denen Detailinformationen, wie z.B. Nebenstrassen oder Strassennamen, erst durch die Verwendung der Magic Lens an den gewünschten Orten auf der Karte zur Verfügung gestellt werden.

Allerdings lässt sich das Konzept der Magic Lens nicht direkt auf die in dieser Arbeit eingeführte thumbnailbasierte Full-Text Visualisierung übertragen, da die Linse aufgrund der Metapher nur den Bereich detailliert darstellt, den der Betrachter durch die Linse sehen kann. Bezogen auf einen Text würden auch die Zeilen, die sich über und unter der durch die Linse fokussierten Zeile befinden, detailliert dargestellt. Diese Zeilen stehen aber in der Regel in keinem Zusammenhang zu dem zentral fokussierten Token - vielmehr erscheint es für den Anwender interessant, die Tokens der aktuellen Zeile vor, innerhalb und nach dem fokussierten Token als Detailinformation dargestellt zu bekommen.

Daher bietet es sich an, den Thumbnails eine Tooltip Funktionalität hinzuzufügen. Sie hat den Vorteil gegenüber der Magic Lens, dass Zusatzinformationen automatisch, also ohne Auswahl einen Operators, dem Betrachter zur Verfügung gestellt werden, wenn er eine gewissen Zeit mit dem Mauszeiger auf einem Objekt verweilt. Da die Thumbnails intern als einfache Rechtecke dargestellt werden, muss über die Mauszeigerposition errechnet werden, welches Token sich aktuell unter dem Mauszeiger befindet. Anhand von Abbildung (7.12) soll das Vorgehen verdeutlicht werden.

Abbildung 7.12: TreeMap Rechteck mit Thumbnail Visualisierung - Tooltip

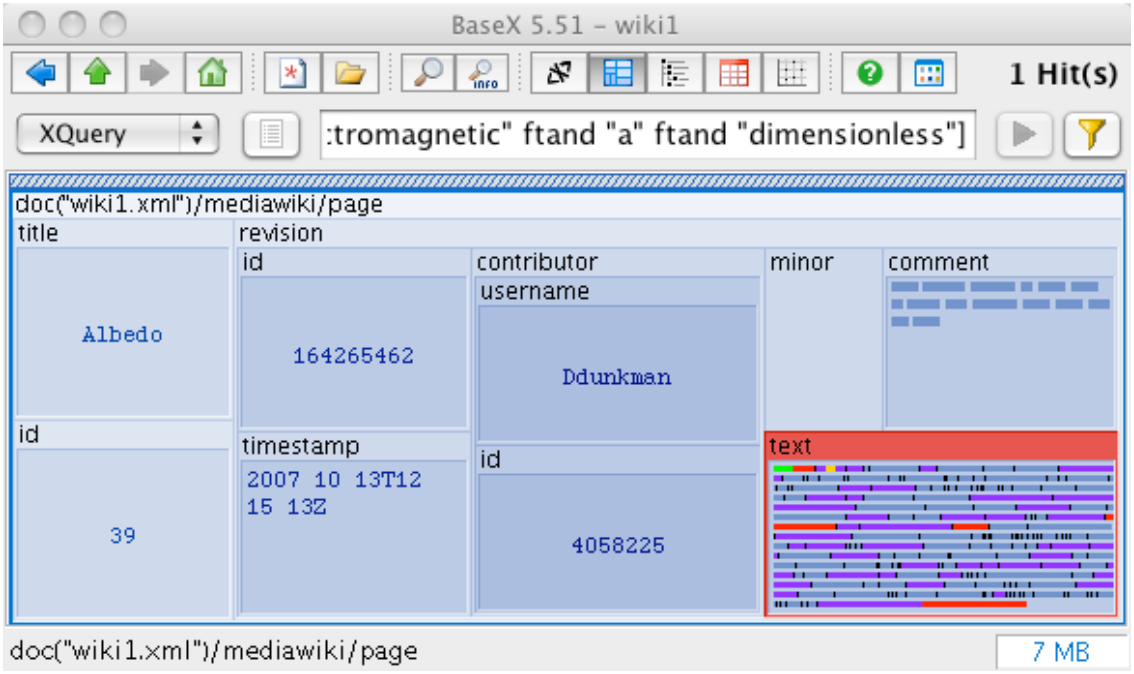


Die nicht ausgefüllten Rechtecke in Abbildung (7.12) dienen zur Verdeutlichung des Vorgehens und können vom Anwender optional zugeschaltet werden. Sie zeigen die einzelnen Token-Thumbnails, aus denen sich der dargestellte Satz-Thumbnail zusammensetzt. Nun kann von der linken oberen Ecke ausgehend durch Aneinanderreihen der Token-Thumbnails ermittelt werden, welches Token sich unter der aktuellen Mauszeigerposition befindet. Ist das Token gefunden - in Abbildung (7.12) ist dies `electromagnetic`, kann es in dem Tooltip angezeigt werden. Wie bereits motiviert, sollen die dem aktuell angewählten Token voran

und nachfolgenden Tokens ebenfalls in den Tooltip aufgenommen werden, um dem Anwender eine Zusatzinformation zur Verfügung zu stellen. Hier ist dies der Textausschnitt `.. ratio of reflected to incident electromagnetic radiation. It is a Dimensionless number ..`, der mit `..` sowohl beginnt wie auch endet um anzudeuten, dass sich der Text vor bzw. nach dem durch den Tooltip lesbaren Bereich fortsetzt. Das dem aktuell angewählten Thumbnail-Token entsprechende Token aus dem Full-Text des Elements wird durch Unterstreichen in dem Tooltip hervorgehoben. Es ist sehr gut zu erkennen, dass dem nachfolgenden Token (`radiation`) ein Satzzeichen folgt (schwarzes Rechteck in dem Thumbnail), welches auch in dem Tooltip zu sehen ist.

Durch den Einsatz der Tooltip Technologie und der Darstellung des voran beziehungsweise nachfolgenden Textes des angewählten Token entsteht dem Betrachter ein informationeller Mehrwert, der so mit einer Magic Lens nicht erreicht wird. Durch sequentielles Überlaufen des Thumbnails ist es dem Betrachter möglich, den visualisierten Text in seiner Gesamtheit zu lesen. Die Verwendung von Thumbnails ermöglicht die platzsparende Visualisierung lange Texter auf wenig Darstellungsfläche.

Abbildung 7.13: TreeMap mit Thumbnail Visualisierung



Die Abbildung (7.13) visualisiert in der TreeMap den einzigen Trefferknoten der nachstehenden Anfrage in der 1 MB Wikipedia Instanz.

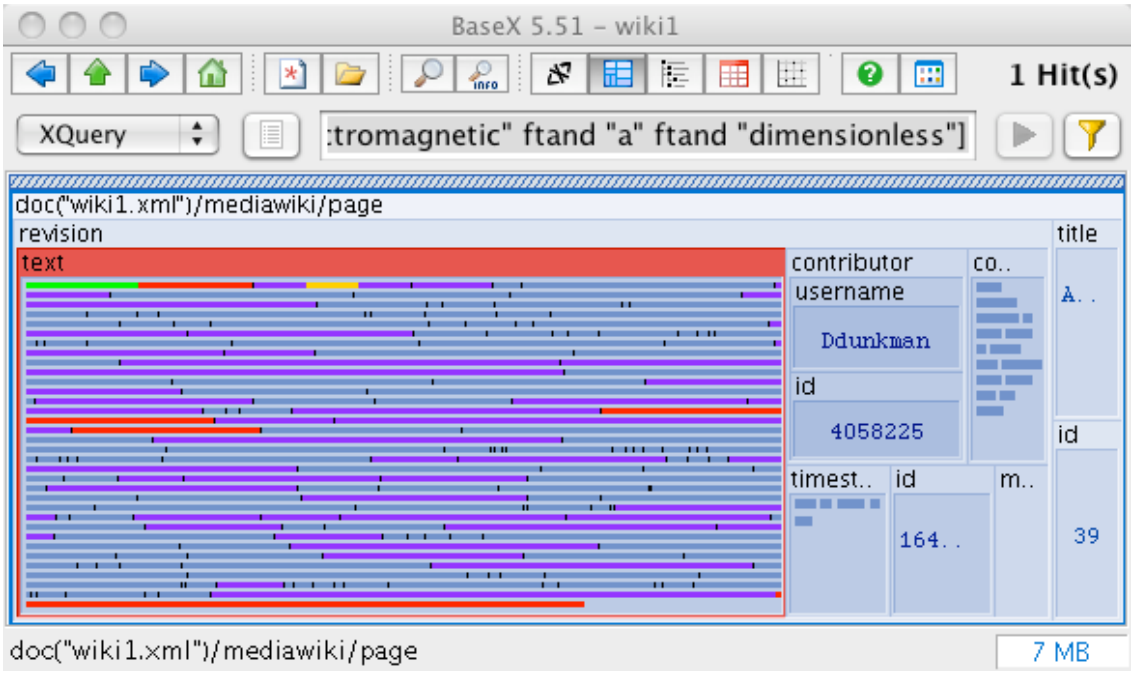
```
/mediawiki/page/revision/text[text() ftcontains "asthe" ftand
    "electromagnetic" ftand "a"  ftand "dimensionless"]
```

Die Thumbnail Visualisierung in dem `text` Element zeigt sehr schön die Verteilung der Treffer in dem Text. Es ist offensichtlich, dass der Suchbegriff, dessen Treffer lila hervorge-

hoben sind, in dem Text häufiger vorkommt, hingegen die gelb und grün hervorgehobenen Einzeltreffer sind. Das Token, das durch die rot eingefärbten Thumbnails repräsentiert wird, kommt hingegen dreimal in dem Text vor.

Die TreeMap in Abbildung (7.13) ist anhand des einfachen Split Layout Algorithmus erzeugt worden, der bereits in Abschnitt (7.1) eingeführt wurde und keine Zusatzinformationen bei der Berechnung der TreeMap verwendet. Dem TreeMap Rechteck mit dem sehr langen Text wird hierbei gleichviel Platz zugewiesen wie seinen Nachbarn, deren Inhalte jeweils nur aus wenigen Zeichen bestehen. Daher wird die zur Verfügung stehende Darstellungsfläche in diesem Fall nicht optimal genutzt. Findet jedoch der Squarified Layout Algorithmus, bei dem die Seitenverhältnisse der TreeMap Rechtecke nahezu quadratisch gewählt werden, Anwendung und gewinnt darüber hinaus die Textlänge an Einfluss auf das Layout, so wird die zur Verfügung stehende Darstellungsfläche erheblich besser genutzt, wie Abbildung (7.14) zeigt [6].

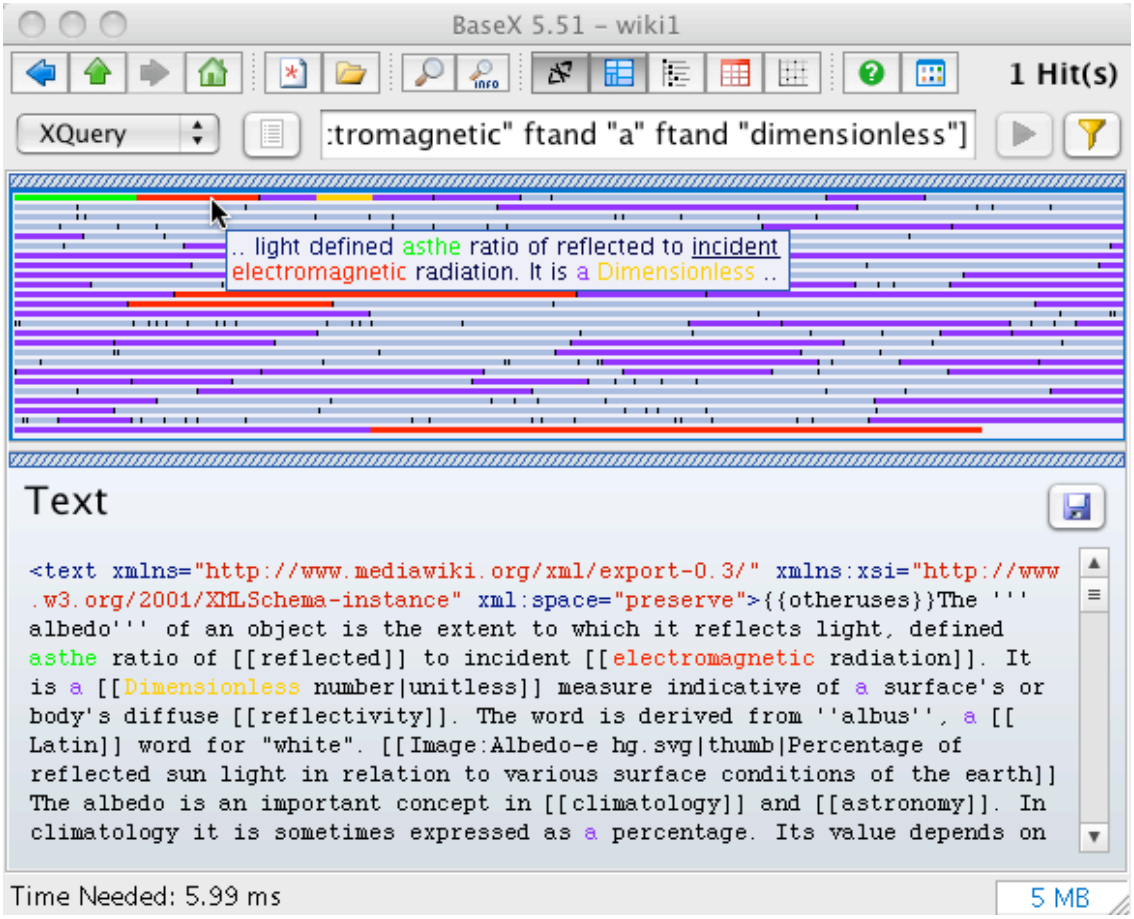
Abbildung 7.14: TreeMap mit Thumbnail Visualisierung - squarified Layout unter Berücksichtigung der Textlänge



Eine Verbindung zwischen den Suchbegriffen aus der Anfrage und den eingefärbten Thumbnails baut der eingeführte Tooltipmechanismus auf. Des Weiteren kann die Einbeziehung einer zusätzlichen Darstellungskomponente, die den Full-Text des Knotens zeigt, verwendet werden. In der Abbildung (7.15) ist diese angepasste Darstellung veranschaulicht.

Der Tooltip zeigt sehr schön, welche Farbe den Suchbegriffen `asthe`, `electromagnetic`, `a` und `dimensionless` aus der Anfrage zugeordnet ist. Bei der Interpretation des Suchergebnisses fällt auf, dass `asthe` und `electromagnetic` in dem gleichen Satz jeweils

Abbildung 7.15: TreeMap Rechteck mit Thumbnail Visualisierung



einmal vorkommen. Darüber hinaus ist aus der Reihenfolge der Rechtecke ersichtlich, dass `asthe` vor `electromagnetic` auftritt. Augenscheinlich ist auch, dass in dem zweiten Satz `dimensionless` von zwei `a` Treffern eingeschlossen ist, und dass im letzten Satz ein `a` vor einem `electromagnetic` zu finden ist.

Bei genauerer Betrachtung des Tooltips fällt auf, dass der Text nicht exakt dem original Knotentext entspricht, z.B. zeigt der Tooltip die das Token `reflected` umschließenden Klammern nicht. Dies ist auf die Bereinigung des Full-Textes von Sonderzeichen vor der Visualisierung in der TreeMap zurück zu führen, um die dort nur beschränkt zur Verfügung stehende Darstellungsfläche mit rein textueller Information optimal zu nutzten.

KAPITEL 8

Schlussbetrachtung und Ausblick

In der vorliegenden Arbeit ist die Implementierung und Optimierung einer baumbasierten Compressed Trie Struktur gezeigt worden, die sehr gut bei exakter und wildcardbasierter Suche funktioniert. Speziell für die fehlertolerante Suche wurde eine weitere tabellenbasierte Index-Struktur eingeführt. Diese Index-Strukturen fungieren als Grundlage des nachfolgend vorgestellten Ansatzes zur Verarbeitung von XQuery Full-Text Anfragen.

Der ausgewählte Ansatz zur Anbindung der Full-Text Index-Struktur an einen vorhandenen XQuery Prozessor ist sehr flexibel und ermöglicht Indexzugriffe ohne deren explizite Spezifikation. Durch die Unterscheidung der drei Verarbeitungsmodi sequentiell, indexbasiert und sequentiell indexbasiert ist sichergestellt, dass jegliche, der Grammatik entsprechende Anfragen verarbeitet werden und, soweit möglich, eine Full-Text Index-Struktur zur Beschleunigung der Anfrageverarbeitung eingebunden wird. Damit entfällt die Indexierung spezieller Pfade. Auch das Einführen spezieller Index-Funktionen, aufgrund derer ein Indexzugriff initiiert wird, kann so umgangen werden. Durch die Anbindung von Full-Text Index-Strukturen konnte erwartungsgemäß die Performance der Anfrageverarbeitung deutlich verbessert werden. Darüber hinaus kann unter Zuhilfenahme statistischer Daten automatisch entschieden werden, ob eine Indexanbindung zur Anfrageverarbeitung sinnvoll ist. Es ist denkbar, zukünftig weitere Index-Strukturen zu ergänzen, die auf spezielle Einsatzbereiche hin optimiert sind, wie z.B. die Suche mit Wildcards oder die Verwendung von Stemming bei der Suche.

Die gezeigte iterative Implementierung der FTOperatoren knüpft an das iterative XQuery Konzept an und setzt es konsequent fort. Darüber hinaus ist die Datenhaltung in den Full-Text Index-Strukturen diesem Konzept angepasst worden, so dass die darin gehaltenen Full-Text Daten iterativ ausgelesen werden können. Für das iterative Zusammenführen von Full-Text Daten, die verschiedenen Suchbegriffen angehören, z.B. bei der wildcardbasierten Suche, ist der Aufbau von Index-Iteratoren-Bäumen eingeführt worden. Sie führen das iterative XQuery Konzept bis zum Auslesen der Full-Text Daten aus der Index-Struktur fort. Das Optimierungspotential für Anfragen, die nur die ersten k-Treffer referenzieren, ist bereits aufgezeigt worden und zukünftig für weitere Anfragetypen zu erproben.

Darüber hinaus ist eine kostenbasierte Auswertungsstrategie für Pfadausdrücke mit

mehreren Prädikaten eingeführt worden, bei der das Prädikat mit der geringsten Treffermenge durch Pfadumkehrung und unter Anbindung der Index-Struktur evaluiert wird. Die weiteren Prädikate können anschließend ohne Indexzugriff berechnet werden. Alternativ ist die Zusammenfassung von Prädikaten unter Zuhilfenahme von Zusatzinformationen aufgezeigt worden. Es wurde ein Entscheidungsverfahren vorgestellt, das vor Evaluation der Anfrage sicherstellt, dass die Zusammenfassung möglich ist.

In zukünftige Arbeiten können die vorgestellten Ansätze und Optimierungen auf FLOWR-Expressions übertragen werden. Da deren Inhalt und Struktur aufgrund der Komplexität der XQuery Grammatik äußerst flexibel ist, übersteigt eine Betrachtung den Rahmen dieser Arbeit und kann als Gegenstand zukünftiger Arbeiten detailliert betrachtet werden.

Abschließend ist ein Visualisierungsverfahren eingeführt worden, bei dem die hierarchische TreeMap zur Visualisierung von XML-Dokumenten mit einer Thumbnail Visualisierung für Texte kombiniert wurde. Hierbei hat sich gezeigt, dass durch den Einsatz der Thumbnailrepräsentatoren auch lange Texte auf einer vergleichsweise geringen Darstellungsfläche darstellen lassen und der visuelle Überblick über die Struktur des Textes erhalten bleibt. Durch die dynamische Anpassung der Thumbnailparameter und der dargestellten Abstraktionsebene kann die Visualisierung effizient berechnet werden, so dass die zur Verfügung stehende Darstellungsfläche optimal genutzt und gleichzeitig der gesamte Text dargestellt wird. Der Verlust an textuellen Information wird durch den Einsatz der Tooltip Technologie kompensiert und bietet darüber hinaus weitere kontextbezogene Informationen. Die gleichzeitige Visualisierung aller Full-Text Treffer und Trefferknoten in der TreeMap bietet Anwendern einen informationellen Mehrwert, den eine textbasierte Darstellung nicht bieten kann. Zukünftig ist es denkbar, weitere Informationen aus den Full-Text Anfragen (z.B. Art der logischen FTOperatoren oder der FTSelections) in die Visualisierung mit aufzunehmen und so dem Betrachter visuell zugänglich zu machen.

Literaturverzeichnis

[1] A.V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers - Priciples, Techniques & Tools*. Seiten 40-85, Second Edition, Pearson International Edition, 2006. (Zitiert auf Seite 21)

[2] S. Amer-Yahia, C. Botev, et al. *XQuery 1.0 and XPath 2.0 Full-Text*. Candidate Recommendation, World Wide Web Consortium, 2008. (Zitiert auf Seiten 1 und 3)

[3] J. Aoe, K. Morimoto. *An Efficient Implementation of Trie Structures*. Software-Practice and Experience, Vol. 22, Pages: 695-721, 1992. (Zitiert auf Seite 16)

[4] S. Boag, D. Chamberlin, et al. *XQuery 1.0: An XML Query Language*. Recommendation, World Wide Web Consortium, 2007. (Zitiert auf Seiten 1 und 3)

[5] T. Bray, J. Paoli, at el. *Extensible Markup Language (XML) 1.0 (Third Edition)*. Technical report, World Wide Web Consortium, 2004. (Zitiert auf Seite 1)

[6] M. Burls, K Huzing, et al. *Squarified treemaps*. In Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization, 2000. (Zitiert auf Seite 69)

[7] S. Ceri, G. Pelagatti. *Statistical profile estimation in database systems*. ACM Computing Surveys, Vol. 20, Pages:191–221, 1988. (Zitiert auf Seite 47)

[8] E. Fredkin. *Trie Memory*. Communications of the ACM, Vol. 3, Pages: 490-500, 1960. (Zitiert auf Seite 16)

[9] M. T. Goodrich, R. Tamassia. *Data Structures and Algorithm in Java*. Seiten: 341-355, 382-392. Second Edition, John Wiley & Soons, 1997. (Zitiert auf Seiten 11, 12 und 60)

[10] P. Griffiths Selinger, M. M. Astrahan, et al. *Access path selection in a relational database management system*. In Proceedings of the ACM SIGMOD, 1979. (Zitiert auf Seite 23)

[11] M. Grinev, A. Fomichev, S. Kuznetsov. *sedna: A Native XML DBMS*. In Proceedings of SOFSEM, 2006. (Zitiert auf Seite 29)

[12] C. Grün. *Pushing XML Main Memory Databases to their Limits*. In GI-Workshop, 2006. (Zitiert auf Seite 1)

[13] C. Grün, Holupirek, et al. *BaseX & DeepFS Join Storage for Filesystem and Database*. In Proceedings of the EDBT, 2009. (Zitiert auf Seiten 10 und 54)

[14] T. Grust. *Accelerating XPath location steps*. In Proceedings of the ACM SIGMOD, 2002. (Zitiert auf Seite 10)

[15] M.Heilig, M. Demarmels, et al. *MedioVis–Visual Information Seeking in Digital Libraries*. In Proceedings of the International Working Conference on Advanced Visual Interfaces, 2008. (Zitiert auf Seiten 8 und 16)

[16] J. Imhof. *Evaluation Strategies for XQuery Full-Text*. Master Thesis, 2008. (Zitiert auf Seite 15)

[17] B. Johnson, B. Shneiderman. *Tree maps: A space-filling approach to the visualization of hierarchical information structures*. In Proceedings of the 2nd International IEEE Visualization Conference, Pages: 284–291, 1991. (Zitiert auf Seite 54)

[18] K. Kaugars. *A Hierarchical Approch to Detail + Context Views*. Unpublished Doctoral Dissertation, New Mexico State University. (Zitiert auf Seite 57)

[19] W. Meier. *eXist: An Open Source Native XML Database*. http://exist.sourceforge.net/indexing.html (Zitiert auf Seite 29)

[20] D. R. Morrison. *PATRICIA Practical Algorithm To Retrieve Information Coded in Alphanumeric*. Journal of the ACM, Vol. 15, Pages: 514 - 534, 1968. (Zitiert auf Seite 16)

[21] W. Odgen, M. Davis, et al. *Documents thumbnail visualization for rapid relevance judgments: When do they pay off?*. In Proceedings at the Seventh Text Retrieval Conference (TREC7), 1998. (Zitiert auf Seite 57)

[22] P. O'Neil, E. O'Neil, et al. *ORDPATHs: Inserting-Friendly XML Node Labels*. In Proceedings at the ACM SIGMAD, 2004. (Zitiert auf Seite 9)

[23] G. G. Robertson, J. D. Mackinlay, et al. *Cone Trees: Animated 3D Visualizations of Hierarchical Information*. In Proceedings of the SIGCHI conference on Human factors in computing systems, Pages: 189 - 194, 1991. (Zitiert auf Seite 53)

[24] R. Spence. *Information Visualization*. Seiten: 149-153, Addison-Wesley, 2001. (Zitiert auf Seiten 53 und 54)

[25] M. Stone, K. Fishkin, et al. *The Movable Filter as a User Interface Tool* In Proceedings of the SIGCHI conference on Human factors in computing systems, Pages: 306 - 312, 1994. (Zitiert auf Seite 66)

Abbildungsverzeichnis

Tabellenverzeichnis

ANHANG A

XML-Full-Text Daten

Auszug Bibliotheksdaten [797 MB]

```
<LIBRARY>
   <MEDIUM mv_id="116658" bib_id="9415">
      <TYP>Buch</TYP>
      <PER>Marckwardt, Albert H.</PER>
      <TIT>Introduction to the English language</TIT>
      <DES>Albert H. Marckwardt</DES>
      <TOW>Oxford</TOW>
      <PUB>Univ. Press</PUB>
      <YEA>1946</YEA>
      <FRM>347 S.</FRM>
      <SIG>6 fse 322/m17</SIG>
      <SEC>Englisch/Fremdsprachenlernen</SEC>
   </MEDIUM>
   ...
</LIBRARY>
```

Tag Index

```
Entries: 21

TAG FREQUENCY, TYP, AVG. CHARS
SIG 1868118x, strings, 14.69
MEDIUM 1855308x, 5.3
TYP 1854968x, 20 values, 4.96
TIT 1781327x, strings, 40.13
SEC 1667810x, strings, 18.25
YEA 1630773x, numeric(89-3003), 3.99
PER 1607536x, strings, 18.24
LAN 1496040x, strings, 4.31
DES 1464941x, strings, 31.8
FRM 1385194x, strings, 14.01
TOW 1308912x, strings, 10.83
PUB 1255706x, strings, 13.87
...
```

Full-Text Index

```
Size on Disk: 329 MB
Entries: 1308637

TOKEN FREQUENCY
buch 1677838
s 1484147
ff 1167891
bvb 956301
720 887280
dt 849783
isbn 780201
3 603007
a 560854
und 548382
...
```

Auszug Wikipedia Enzyklopädie [100 MB]

```
<mediawiki>
   ...
   <page>
      <title>Autism</title>
      <id>25</id>
      <revision>
         <contributor>
            <username>Eubulides</username>
            <id>3573537</id>
         </contributor>
         <text>"Autism"is a [[Neurodevelopmental disorders|brain development disorder]]
         characterized by impairments in social interaction and communication,
         and restricted and repetitive behavior, all exhibited before a child is three
         years old. These characteristics distinguish autism from milder
         [[autism spectrum disorder]]s (ASD). ...
         </text>
      </revision>
   </page>
   ...
</mediawiki>
```

Tag Index

```
Entries: 20

TAG FREQUENCY, TYP, AVG. CHARS
id 375365x, numeric(1-165998412), 6.77
page 131833x, 3.5
title 131833x, strings, 17.28
revision 131833x, 5.67
timestamp 131833x, strings, 20.0
contributor 131833x, 7.3
text 131833x, strings, 7498.68
comment 114503x, strings, 49.05
username 111699x, strings, 8.79
...
```

Full-Text Index

```
Size on Disk: 900 MB
Entries: 2237863

TOKEN FREQUENCY
the 7826675
of 4809229
and 3114488
in 2727439
a 2269684
to 2106863
is 1182161
was 1155308
s 1147818
for 1002427
...
```