Universität Rostock
Lehrstuhl Datenbank- und Informationssysteme


Diplomarbeit

# Indexierung lokaler Daten in Peer-to-Peer-Netzwerken

**Clemens Nafe**

geb. am 05. Oktober 1979 in Schwerin (Meckl.)
Matrikelnummer 099201956

| | |
|---|---|
| Betreuer: | Dr. Holger Meyer, Dipl.-Inf. Andre Zeitz |
| Erster Gutachter: | Prof. Dr. Andreas Heuer |
| Zweiter Gutachter: | Prof. Dr. Adelinde Uhrmacher |

eingereicht am 31. März 2005

# Zusammenfassung

In dieser Diplomarbeit wurde ein Verfahren entwickelt um in dezentralisierten, unstrukturierten Peer-to-Peer Overlay Netzwerken Abschätzungen treffen zu können, welcher Netzteilnehmer wahrscheinlich die besten Ergebnisse auf eine Anfrage liefern wird. Es werden verschiedene Arten der Abstraktion von Textdaten, deren Indexierung und deren Propagierung im Netz vorgestellt. Um zu validieren, ob diese Betrachtungen zur Senkung des Netzwerkverkehrs führen, ohne dabei eine signifikante Verschlechterung der Ergebnisqualität zu verursachen, wurde ein Prototyp entwickelt.

# Schlüsselwörter

dezentralisiert, unstrukturiert, Peer-to-Peer Overlay Netzwerk, Indexierung, Routing, Histogramme, Hashfunktionen

# Abstract

In this thesis a procedure for decentralized, unstructured peer-to-peer overlay networks was developed to be able to make estimations which network host will probably supply the best results on a query request. Different kinds of textdata abstraction are presented as well as their indexation and propagation in the net. In order to validate whether these views lead to the reduction of network traffic without causing thereby a significant degradation of the result quality a prototype was developed.

# Keywords

decentralized, unstructured, peer-to-peer overlay network, indexing, routing, histograms, hashing

# CR-Klassifikation

C.2.1 Network communications
C.2.2 Routing protocols
D.2.8 Performance measures
D.2.11 Data abstraction
D.4.4 Network communication
H.2.4 Distributed databases
H.3.1 Abstracting methods, Indexing methods
H.3.4 Distributed systems

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Hintergrund

So alt wie das Internet ist auch das Client/Server-Prinzip. Server sind Dienstleister, die Dienste und Dateien anbieten. Clients sind Kunden, die Dienste in Anspruch nehmen bzw. Dateien anfordern. Dieses Prinzip funktioniert solange einwandfrei, bis die Bedeutung eines Servers unnatürlich hoch gestiegen ist. Je höher die Bedeutung eines Servers, desto gefragter sind die von ihm angebotenen Dienste oder Dateien. Server mit oft benötigten Diensten können auch als Hotspot-Server bezeichnet werden. Im schlimmsten Fall wollen so viele Clients gleichzeitig die vom Server angebotenen Dienste in Anspruch nehmen, dass der Server überlastet und nicht mehr erreichbar ist. Im Gegensatz dazu kann die Bedeutung eines Servers soweit sinken, dass kein Client über einen längeren Zeitraum hinweg irgendeinen Dienst des Servers in Anspruch nehmen möchte. Von Nachteil sind beide Szenarien, denn entweder ist der Server so überlastet, dass er die ihm zugeteilte Aufgabe nicht adäquat erledigen kann, oder er bietet unnütze Dienste an und verbraucht trotzdem Strom und somit Geld. Im letzteren Fall kann man sagen, dass jeder nicht genutzte Server auch nicht genutzte Rechenkapazität bedeutet.

Eine Lösung, um sowohl Hotspot-Server als auch verschwendete Rechenkapazität zu vermeiden, findet sich in Peer-to-Peer (P2P)-Netzen. P2P-Netze sind Computerverbünde, in denen das klassische Client/Server Prinzip aufgebrochen ist. Ein Rechner kann sowohl Client als auch Server sein. Hotspot-Server können umgangen werden, indem stark gefragte Dienste und Dateien auf mehr als nur einem Rechner angeboten werden. Auf diese Art wird die Anfragelast aufgeteilt. Vorteil dieser Vorgehensweise ist, dass die Wahrscheinlichkeit vom Auftreten überlasteter Rechner drastisch reduziert wird und dass anfragende Rechner (Clients) mit einer kürzeren Reaktions- und Downloadzeit rechnen können. Verschwendete Rechenkapazität wird auf die gleiche Art vermieden.

Seit 1999 ist ein explosionsartiger Anstieg von verschiedensten P2P-Netzen zu beobachten, welche sich in drei Klassen einteilen lassen [LCC+02].

- **zentralisiert (semi-dezentralisiert):** Die Haupteigenschaft dieser P2P-Netze ist, dass es einen dedizierten Index-Rechner gibt. Dieser Index-Rechner hält eine ständig aktualisierte, globale Wissensdatenbank. Der Index-Rechner nimmt nach dem Client/Server-Prinzip ganz eindeutig die Rolle des Servers ein. Der Unterschied zum Client /Server-Prinzip liegt in der Rolle aller anderen Netzteilnehmer. Diese Netzteilnehmer sind sowohl Client als auch Server. Sie selber besitzen kein Wissen über die im Netz vorhandenen Dateien. Alle Anfragen werden an den Index-Rechner gestellt. Dieser durchsucht seine Datenbank und gibt alle Vorkommen des Gesuchten im Netz zurück. Nun wenden sich die Teilnehmer direkt an den Halter der Information. Außer den Indexabfragen wird jede weitere Kommunikation an dem Index-Rechner vorbei geführt (siehe Abbildung 1.1). Die Vorteile dieser Netze liegen in dem einfachen Eintritt in das Netz, dem einfachen Finden von Informationen und dem Finden von Informationen trotz der Abwesenheit des Halters. Nachteil dieses Netzes ist die Abhängigkeit von zentralen Rechnern. Fällt der Index-Rechner aus, ist das Netz sozusagen inexistent. Als Beispiel für solch ein Netz kann die ursprüngliche Version des Napster Projektes betrachtet werden [Hof02, Rö03].

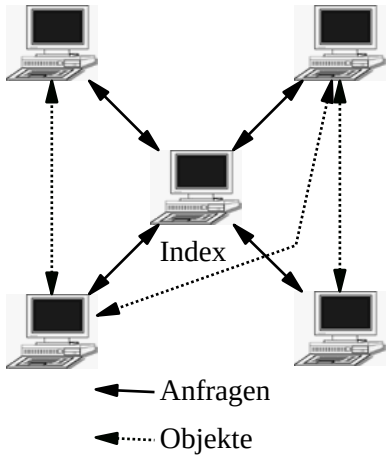



**Abbildung 1.1:** Struktur eines zentralisierten P2P-Netzes: Alle Anfragen werden an den zentralen Server gestellt und von diesem beantwortet. Die eigentlichen Daten werden direkt von den Haltern bezogen.

Es existieren verschiedene Versionen von Netzen, welche versuchen die Vorteile besser zu nutzen und die Nachteile in den Hintergrund treten zu lassen. Eine Art solcher Netze unterscheidet sich dadurch, dass es mehrere Index-Rechner (Supernodes) gibt, die untereinander

zu einem eigenen Netz zusammengefasst sind (siehe Abbildung 1.2). Normale Teilnehmer befinden sich nur in einem Teil des Gesamtnetzes und können bei Bedarf in einen anderen wechseln, indem sie sich mit einem anderen Supernode verbinden. Diese Art von Netzen ist weniger anfällig für Netzausfälle, da jeder Netzteilnehmer in der Lage ist, die Rolle eines Supernodes zu übernehmen. Beispiel für diese Art von Netzen sind alle P2P-Netze, die auf dem FastTrack-Protokoll basieren [Swg05].

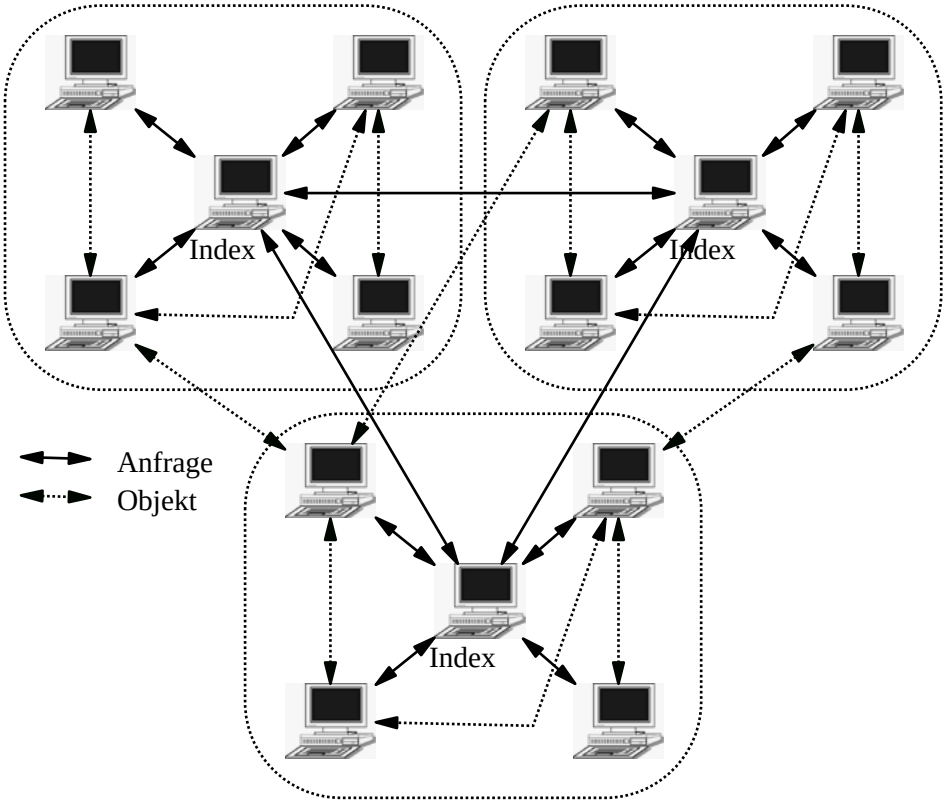



**Abbildung 1.2:** Struktur eines FastTrack-Netzes: Supernodes fungieren als lokale Index-Server, leiten Anfragen aber auch an weitere Index-Server weiter. Dies erhöht die Skalierbarkeit und erniedrigt die Ausfallwahrscheinlichkeit des Netzes.

- **dezentralisiert, strukturiert:** Diese Netze zeichnen sich dadurch aus, dass sie keine zentralen Server besitzen, die Netzwerktopologie aber bis zu einem bestimmten Grad festgelegt ist. Jeder Netzteilnehmer ist mit einer Menge von anderen Teilnehmern verbunden, wobei die Menge sehr viel kleiner sein kann als die Gesamtanzahl an Teilnehmern im Netz. Die Liste aller Nachbarn eines Teilnehmers wird für ge-

wöhnlich in einer Hash-Tabelle gespeichert. Hierbei ist ein Nachbar, wer direkt und nicht über Dritte mit dem Teinehmer verbunden ist. Teilnehmer können allerdings jederzeit Verbindungen zu ihren Nachbarn lösen und neue Teilnehmer in ihre Nachbarschaft aufnehmen. Die Gründe hierfür sind für gewöhnlich abhängig von der Definition einer guten Nachbarschaft. Ein guter Nachbar kann ein Teilnehmer mit möglichst geringer Antwortzeit sein, einer mit möglichst hoher Lebenszeit (Uptime) oder einer mit Daten, die den eigenen ähnlich sind. Auf Grund verschiedenster Replikationsstrategien sind die Daten nicht zwangsläufig zufällig auf bestimmten Teilnehmern vorhanden. Diese sollen das Auffinden von Informationen schneller und effizienter machen [LCC+02, Poh04]. Einige Vertreter dieser Art von Netzen sind Chord, CAN und Tapestry [SMK+01, RFH+01, ZKJ01].

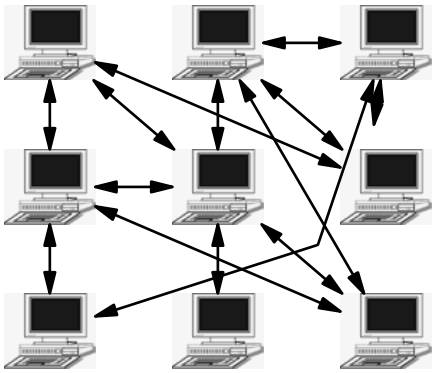


**Abbildung 1.3:** Beispiel für ein dezentralisiertes P2P-Netz

- **dezentralisiert, unstrukturiert:** Diese Netze besitzen weder zentrale Server noch irgendeine vorgegebene Netzwerkstruktur. Das komplette Wissen der Netzwerkteilnehmer (Hosts) besteht aus dem eigenen Repositorium und welcher andere Host direkter Nachbar ist. Da kein Wissen über den Inhalt anderer Repositorien vorhanden ist, werden Anfragen mittels flooding an alle erreichbaren Hosts gesandt. Der Netzwerkverkehr, der dabei entsteht, ist beträchtlich. Vorteil dieses Netzes ist dessen Robustheit gegenüber Ein- und Austritten aus dem Netz. Der Administrationsaufwand, der dabei entsteht, ist völlig unerheblich. Die Probleme bestehen im Auffinden von Hosts bei Netzeintritt sowie der hohen Netzbelastung. Viele Vertreter dieser Netze basieren auf dem Gnutella-Protokoll [Cli].

Ob der Robustheit und der Unabhängigkeit von zentralen Indexstrukturen eignen sich dezentralisierte P2P-Netze besonders gut für Netzwerke, denen Ausfall-

sicherheit und eine sich stark verändernde Hostzahl abverlangt wird. Der Fokus dieser Arbeit liegt auf unstrukturierten Netzen, genauer auf der Problematik der Netzlast, die bei Anfragen entsteht.

## 1.2 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut: Das aktuelle Kapitel beschreibt den Hintergrund dieser Arbeit. Das Kapitel 2 beschäftigt sich mit verschiedenen Indexierungsmethoden für Textdaten. Der Fokus liegt dabei immer auf der Verwendbarkeit für XML-Dokumente. Anschließend werden in Kapitel 3 Verfahren zum Routing von Anfragen in P2P-Netzen erläutert. Designentscheidungen und der konzeptionelle Aufbau eines Prototypen werden im Kapitel 4 besprochen. Kapitel 5 geht auf die konkrete Implementierung und die einzelnen Komponenten des Prototypen ein. Es werden Entscheidungen begründet und Konzepte erklärt. Ergebnisse und Erkenntnisse aus der Arbeit mit dem Prototypen werden in Kapitel 6 vorgestellt. Eine Gesamtzusammenfassung und Erweiterungsmöglichkeiten gibt abschließend das Kapitel 7.

# Kapitel 2

# Indexierung von XML-Dokumenten

In diesem Kapitel werden verschiedene Indexierungsmethoden vorgestellt. Diese sind nicht zwangsläufig für die Indexierung von XML-Dokumenten vorgesehen und lassen sich auf den ersten Blick auch nicht dafür verwenden. Sie sollen dennoch vorgestellt werden, da die allgemeine Vorgehensweise, die hinter diesen Methoden steckt, in gewissem Maße übertragbar ist.

Bei der Indexierung geht es zwar um die lokal vorhandenen Daten, doch ist der Index nicht für den lokalen Gebrauch bestimmt. Auf jedem Host ist ohnehin ein Mechanismus vorhanden um in dem lokalen Repositorium zu suchen. Der Index, welcher in dieser Diplomarbeit besprochen wird, ist einer der auf Fremdrechnern eine Abschätzung über die lokal vorhandenen Daten zulässt. Dies beinhaltet, dass neben lokalen Daten auch externe Daten indexiert werden müssen. Der Fokus liegt hier auf der Abschätzung über Daten fremder Hosts, daher wird der Anteil von lokalen Daten in dem lokalen Index sehr gering sein.

Es gibt keine allgemeingültige Methode, um Daten zu indexieren. Für die Entwicklung einer Anwendung, welche das Indexieren von Daten beinhaltet, ist das Wissen über die zu erwartenden Daten unerlässlich. Nicht alle Daten eignen sich für jede Art der Indexierung. Die erste Frage in diesem Zusammenhang ist die der Schemata. Auf den ersten Blick können Schemata vorhanden sein oder nicht. Oft kann dies so nicht verallgemeinert werden sondern bedarf näherer Betrachtung. Genau genommen lassen sich sechs verschiedene Szenarien kreieren, welche folgend näher beschrieben werden.

- **Szenario** 1**:** *Global-globales Schema* - Es existiert ein einziges globales Schema, welches für alle XML-Dokumente auf allen Hosts im P2P-Netz gültig ist.

- **Szenario** 2**:** *Lokal-globales Schema* - Es gibt kein allgemeingültiges globales Schema, sondern jeder Host hält ein eigenes Schema, welches für alle XML-Dokumente des Hosts gültig ist.

- **Szenario** 3**:** *Viele lokale Schemata* - Jedes Dokument auf jedem Host entspricht seinem eigenen Schema. Dies widerspricht zwar etwas dem Sinn von Schemata, ist aber ein durchaus denkbares Szenario. Einige XML-Retrieval-Systeme wie Tequyla-TX liefern als Ergebnis einer Anfrage immer komplette XML-Dokumente und ein dazugehöriges Schema [CSA+02]. Speichert man diese Ergebnisdokumente in seinem lokalen Repositorium, ist man genau mit diesem Szenario konfrontiert.

- **Szenario** 4**:** *Wenige lokale Schemata* - Auf jedem Host im P2P-Netz gibt es eine Menge von Schemata, die kleiner als die Menge der Dokumente ist. Dieser Fall könnte entstehen, wenn lokal alle Dokumente mit dem gleichen Schema zu einem Ordner zusammengefasst werden. So würde für jeden Ordner ein anderes Schema gelten. Dieses Szenario ist ein Spezialfall des vorangegangenen.

- **Szenario** 5**:** *Einige Schemata* - Es kann auf jedem Host vorkommen, dass sowohl Dokumente mit als auch ohne Schema existieren.

- **Szenario** 6**:** *Keine Schemata* - Es gibt für kein einziges Dokument im gesamten P2P-Netz ein Schema.

Eine weitere Frage ist, ob später die Anfragen die Dokumentstruktur unterstützen sollen, reiner Volltext gesucht wird, oder beides.

## 2.1 Data-Guides

Data-Guides (auch Tree-Guides) sind bekannt aus dem Lightweight Object Repository (Lore) und dienen dem Speichern von Metadaten [MAG+97]. In Lore werden alle Daten als OEM (Object Exchange Model) Graphen abgespeichert [PGMW95]. Data-Guides sind OEM konforme Graphen und werden in Lore auch als solche abgespeichert. Sie geben die Struktur der in der Datenbank abgespeicherten Daten wieder. Auch haben sie die Eigenschaft, niemals Informationen erwarten zu lassen, die in dieser nicht gespeichert sind. Sie verhalten sich wie ein XML-Schema für XML-Dateien. Der Inhalt der Datenbank entspricht einem Data-Guide wie die Struktur eines XML-Dokumentes seinem Schema. Der Unterschied ist, dass das Dokument entsprechend dem Schema aufgebaut ist, wogegen Data-Guides entsprechend dem Inhalt der Datenbank aufgebaut werden.

Um Data-Guides beschreiben zu können, wird zunächst näheres Wissen über OEM-Datenbanken benötigt. Hierfür werden einige OEM Begriffe definiert [GW97].

**Definition 2.1.1 (Beschrifteter Pfad)** *Ein beschrifteter Pfad eines OEM Objektes o ist eine Sequenz von punkt-getrennten Beschriftungen* $l_1.l_2 \ldots l_n$*. Mit*

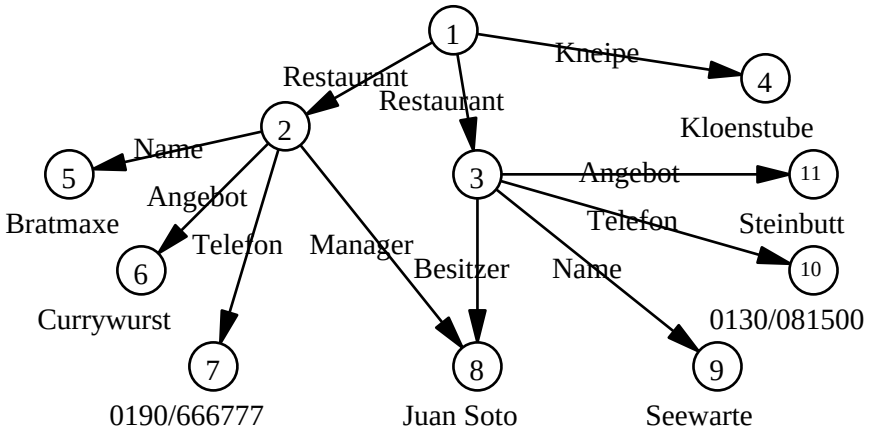



**Abbildung 2.1:** Beispiel einer OEM-Datenbank

*ihnen wird ausgehend von Objekt o ein Pfad über n Kanten* $(e_1, \ldots, e_n)$ *beschrieben. Zu einer Kante* $e_i$ *gehört die Beschriftung* $l_i$*.*

In Abbildung 2.1 stellen zum Beispiel "Restaurant.Name" und "Kneipe" gültige beschriftete Pfade vom Objekt 1 dar.

**Definition 2.1.2 (Datenpfad)** *Ein Datenpfad eines Objektes o ist eine alternierende Sequenz punkt-getrennter Beschriftungen und Objekt-Identifikatoren (OIDs)* $l_1.o_1.l_2.o_2 \ldots l_n.o_n$*. Damit wird, ausgehend von Objekt o, ein Pfad über n Kanten* $(e_1, \ldots, e_n)$ *durch n Objekte* $(x_1, \ldots, x_n)$ *beschrieben. Kante* $e_i$ *hat die Beschriftung* $l_i$*, und das Objekt* $x_i$ *hat die OID* $o_i$*.*

Für Objekt 1 ist ein gültiger Datenpfad "Restaurant.2.Name.5" (Abbildung 2.1).

**Definition 2.1.3 (Instanz eines beschrifteten Pfades)** *Ein Datenpfad d ist eine Instanz eines beschrifteten Pfades b, wenn die Sequenz der Beschriftungen in d und b gleich sind.*

Der Datenpfad "Restaurant.2.Name.5" ist eine Instanz des beschrifteten Pfades "Restaurant.Name".

**Definition 2.1.4 (Zielmenge)** *Die Zielmenge von Objekt s ist eine Menge t von OIDs. Für einen beschrifteten Pfad b gilt dann* $t = \{o | l_1.o_1.l_2.o_2 \ldots l_n.o$ *ist ein Datenpfad und Instanz von* $b\}$

Für den beschrifteten Pfad "Restaurant.Name" ist die Zielmenge somit $\{5, 9\}$.

Nun können Data-Guides formal eingeführt werden.

**Definition 2.1.5 (Data-Guides)** *Ein Data-Guide für eine OEM Datenbank o ist ein OEM Objekt s. Es gilt, dass jeder beschriftete Pfad in o genau eine Datenpfad-Instanz in s hat, und jeder beschriftete Pfad in s existiert auch in o.*

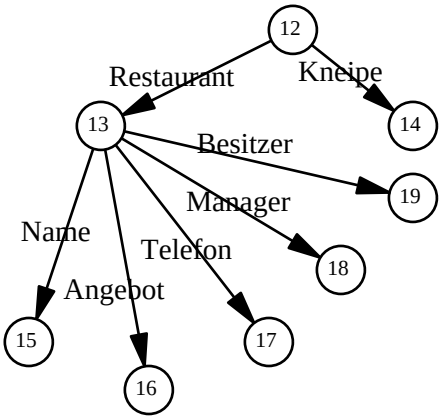



**Abbildung 2.2:** Data-Guide für Abbildung 2.1

Abbildung 2.2 zeigt einen Data-Guide für die OEM Datenbank in Abbildung 2.1. Anhand des Data-Guides kann vor Abarbeitung einer Anfrage überprüft werden, ob sich überhaupt Dokumente im Repositorium befinden, die der Anfrage genügen. Wie weiter oben erwähnt, dienen sie nur dazu Metainformationen zu speichern. Sie speichern nur die Struktur von graphenbasierten Datenspeichern. Auch XML-Dokumente liegen in einer Baumstruktur vor. Ein Element kann Unterelemente haben, die wiederum Unterelemente haben. Wird an einen XML-Datenspeicher eine strukturunterstützende Anfrage gestellt, also eine der Form

```
<xsl:value-of select="/CustomerOrder/CustomerInformation/
 Address[@type='shipping']/City">
```

so kann sofort ein negatives Ergebnis (kein Ergebnis) zurück gegeben werden, wenn der Pfad "/CustomerOrder/CustomerInformation/Address/City" nicht im Data-Guide vorkommt. Volltextanfragen werden dagegen nicht unterstützt.

Auf Seite 7 wurden sechs Szenarien über das Vorhandensein von Schemata aufgezeigt. Für die Data-Guide-Erstellung ist es von Vorteil, wenn die aktuelle Situation den Szenarien "Global-globales Schema" oder "Lokal-globales Schema" entspricht. So kann ohne großen Aufwand ein globaler oder host-globaler Data-Guide erstellt werden. Fehlen solche globalen Schemata, so muss der Data-Guide von einem virtuellen XML-Dokument gebildet werden, wobei das virtuelle Dokument alle Dokumente des Repositoriums beinhaltet.

Es sei noch angemerkt, dass Data-Guides im ungünstigsten Fall nur unwesentlich kleiner sind als das komplette Repositorium. Abhilfe sollen hier minimale Data-Guides oder Strong-Data-Guides schaffen. Strong-Data-Guides sind ein Kompromiss aus Minimalität und Einfügeperformanz.

## 2.2 Tries

Tries sind Bäume zur Speicherung von Wörtern eines Alphabetes [Fre60, HS99a, KM03a]. Dabei müssen Elemente des Alphabetes nicht zwangsläufig nur aus einem Zeichen bestehen. Das Alphabet kann aus chemischen Elementen, Silben, Buchstaben u.s.w. bestehen. Tries kommen aus dem Gebiet des Textretrieval und sollen dazu dienen, in dieser Datenstruktur abgespeicherte Wörter schnell zu finden.

Der Trie-Baum besteht aus zwei Komponenten, den Trie-Knoten und den Datensätzen. Jeder Trie-Knoten besitzt für jedes Element des Alphabetes einen Zeiger. Datensätze beinhalten die zu speichernden Wörter. Es werden solange Trie-Knoten hintereinander verlinkt, bis der Pfad ein abgespeichertes Wort eindeutig identifiziert. Abbildung 2.3 zeigt ein Beispiel-Trie. Dort sieht man, dass die Wörter 'Autofahrt' und 'Autohaus' solange gleich verwiesen werden, bis sich die Wörter unterscheiden. Der Pfad von der Wurzel bis zu einem Datensatz

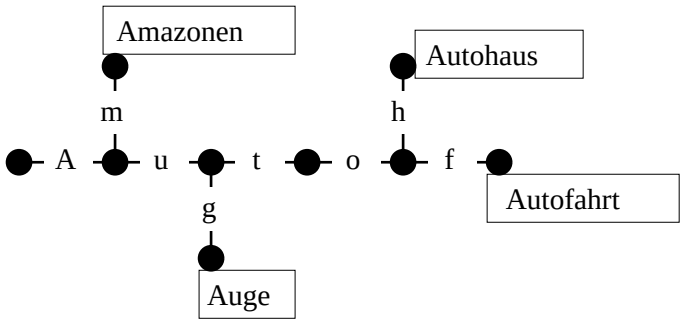
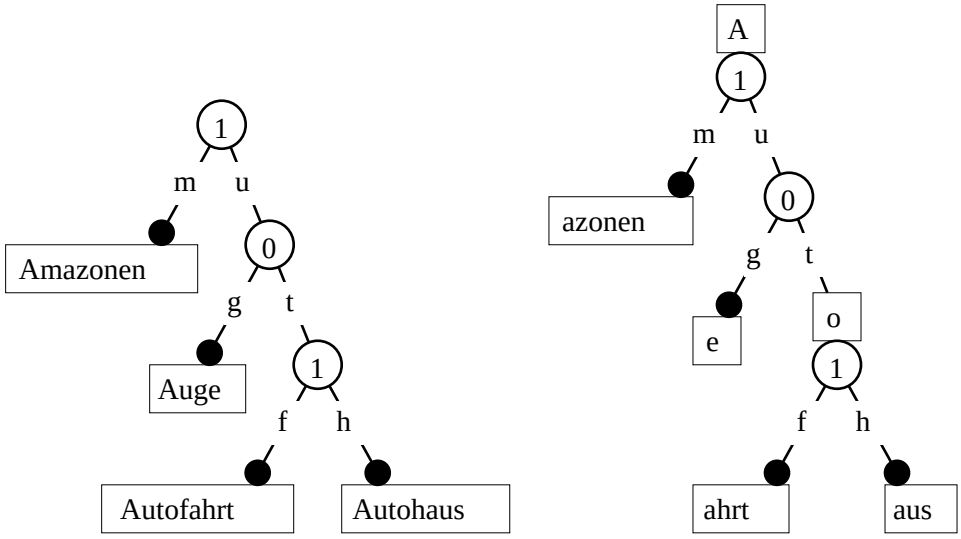



**Abbildung 2.3:** Beispiel-Trie: Um die Übersichtlichkeit zu wahren, sind nur die Verweise eingezeichnet, die nicht in das Leere zeigen.

kann maximal so lang werden wie ein einzufügendes Wort "Buchstaben"[1] hat.

Tries haben haben das Problem, dass sie schnell entarten können. Entartet sind sie dann, wenn sie offensichtlich unbalanciert sind. Fügt man zum Beispiel ausschließlich Wörter ein, die alle gleich beginnen, so haben die Knoten bis zur ersten Verzweigung nur eine Auslastung von $1/|Alphabet|$. Ein ähnliches

[1] Buchstabe meint hier und im weiteren genau genommen ein Element aus dem konkreten Alphabet.




**(a)** PATRICIA-Baum **(b)** Präfix-Baum

**Abbildung 2.4:** PATRICIA-Bäume

Problem ist, dass einige Zeiger grundsätzlich leer bleiben werden. Dies kann an einem realen Beispiel deutlich gemacht werden. Buchstaben in einem englischen Text sind nicht gleichverteilt, sondern unterliegen einer relativ festen statistischen Verteilung. Zum Beispiel sind nur $0,1\%$ der Buschstaben ein 'q', wogegen $10,2\%$ ein 'e' sind. Weniger ein Problem denn unnötiger Rechenaufwand ist, dass oft unnötige Buchstabenvergleiche durchgeführt werden müssen. So wird in Abbildung 2.3 der erste Buchstabe völlig unnötig überprüft.

## 2.3 PATRICIA-Bäume

PATRICIA-Bäume (Practical Algorithm To Retrieval Information Coded In Alphanumeric) (auch PATRICIA-Tries) sind Tries, die einige der Probleme von normalen Tries beheben sollen [Mor68, HS99a, KM03a]. Zum Beispiel fallen hier die unnötigen Vergleiche weg, denn sie speichern in jedem Knoten einen Offset von zu überspringenden Buchstaben oder auch den Index des nächsten relevanten Buchstaben (relative/absolute Adressierung). In Abbildung 2.4(a) sieht man einen PATRICIA-Baum für das Beispiel in Abbildung 2.3.

Aber auch sie bergen noch Probleme. Weil Buchstaben übersprungen werden, kann man nicht davon ausgehen, dass wenn ein Suchwort auf einen Pfad passt, das Wort auch tatsächlich gespeichert ist. Daher muss am Ende eines Pfades nach wie vor das Suchwort mit dem gespeicherten Datensatz verglichen

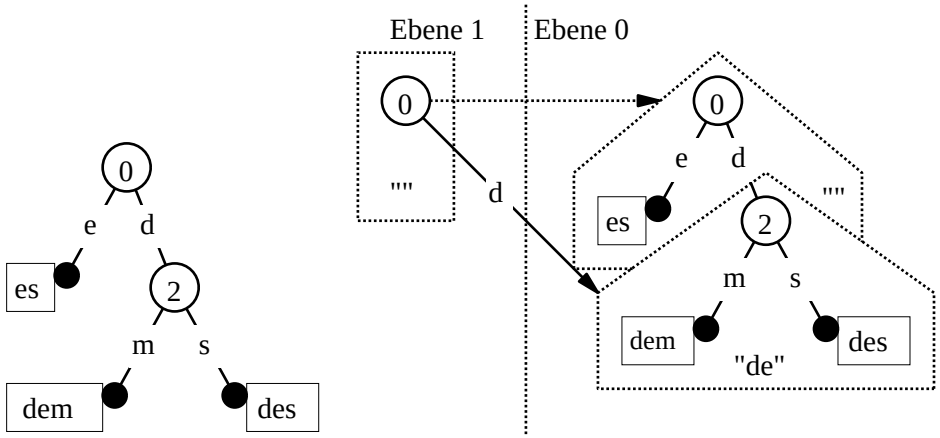
werden. Dies wäre aber manchmal gar nicht notwendig, denn wenn man in dem Beispiel nach dem Wort "Mutation" sucht, stellt man erst bei dem dritten Test fest, dass das Wort nicht enthalten ist. Hätte man aber den ersten Buchstaben überprüft, wäre dies sofort aufgefallen.

Damit dieses Problem nicht mehr auftritt gibt es eine Spezialform des PATRICIA-Baumes (PT-Baum), den Präfix-Baum. Er speichert nicht nur den Offset, sondern auch die übersprungene Buchstabensequenz. Ein Beispiel ist in Abbildung 2.4(b) gezeigt. Um ein Wort im Präfix-Baum zu suchen werden zwar mehr Tests als im PT-Baum benötigt, aber immer noch deutlich weniger als im normalen Trie. Vorteil hierbei ist, dass nun keine Buchstaben mehr übersprungen werden und somit auch nicht mehr der komplette Datensatz abgespeichert werden muss. Einen Pfad von der Wurzel bis zum Blatt zu verfolgen ergibt automatisch das gespeicherte Wort.

## 2.4 Index Fabric

Die Probleme von Tries wurden teilweise mit PT-Bäumen behoben, doch auch diese haben noch Defizite. So kann ein PT-Baum sehr unbalanciert werden. Grundsätzlich gilt, dass die maximale Pfadlänge kleiner gleich der Länge des längsten gespeicherten Wortes und die minimale gleich 1 ist. Beide Fälle können aber auch parallel eintreten. So kann ein Zweig des PT-Baumes sehr kurz und unverzweigt sein, wogegen ein anderer lang und stark verzweigt ist. Die Index Fabric ist eine Indexart, die auf PT-Bäumen aufsetzt, aber nicht die Probleme derer hat [CS01, KM03b]. Index Fabric vereint zwei sonst eher gegenläufige Ziele. Sie unterstützt mehrere verschiedene Zugriffspfade auf gleiche Daten gleichzeitig, ist dabei aber immer noch sehr effizient. Effizient heißt hier, dass der Index äußerst schnell arbeitet und spätestens mit dem dritten Plattenzugriff einen gefundenen Datensatz findet (Erklärung später).

Die Erweiterung zu PT-Bäumen erfolgt in Form einer Aufteilung des Baumes auf Blöcke. Außerdem wird neben der vertikalen Baumstruktur eine horizontale Ebenenhierarchie eingeführt. Diese horizontalen Ebenen erlauben eine ausbalancierte Sicht auf den zugrunde liegenden (unbalancierten) PT-Baum und ein Überspringen mehrerer Knoten in diesem. Abbildung 2.5(a) zeigt einen PT-Baum. Er wird in etwa gleich große Teilbäume zerlegt und auf Blöcke aufgeteilt. In Abbildung 2.5(b) existiert der Originalindex immer noch (Ebene 0), jedoch wurde er auf zwei Blöcke aufgeteilt. Der Knoten in Ebene 1 dient als Index über die Blöcke auf Ebene 0. Ähnlich den Präfix-Bäumen wird in jedem Block das Präfix eines jeden Wortes des Blockes gespeichert. Es gibt zwei Arten von Zeigern: Zeiger innerhalb von Ebenen und zwischen verschiedenen Ebenen. Beide Arten können wiederum in zwei spezielle Zeiger unterteilt werden. Innerhalb von Ebenen verbinden *near Links* Knoten innerhalb der Blöcke und *split Links* Knoten verschiedener Blöcke. Zeiger $e$ in Abbildung 2.5(b) ist somit ein near




**(a)** PATRICIA-Baum **(b)** Index Fabric

**Abbildung 2.5:** Ausbalancieren eines PATRICIA-Baumes

und $d$ (rechts) ein split Link. *Far Links* verbinden einen Vaterknoten mit seinem Kindknoten, welcher sich auf verschiedenen Ebenen ($i+1$ und $i$) befinden. In den Grafiken werden sie als normaler Pfeil dargestellt. Die zweite Unterzeigerart ist der *direct Link*. Er wird als gepunkteter Pfeil dargestellt und verbindet ein und den selben Knoten, nur dass diese sich auf verschiedenen Ebenen ($i+1$ und $i$) befinden. In der Abbildung ist d (links) ein far Link und der gepunktete Pfeil ein direct Link.

Die oberste Ebene (jene mit der höchsten Ebenennummer) dient als Einstiegspunkt in den Index und wird grundsätzlich nur von einem Block belegt. Vergrößert sich der PT-Baum auf Ebene 0 soweit, dass die Anzahl der benötigten Blöcke größer ist, als der Block auf Ebene 1 verlinken kann, so wird dieser auf zwei Blöcke aufgeteilt. Um wiederum einen Einstiegspunkt zu haben, wird eine neue Ebene 2 angelegt. Ein Beispiel hierfür findet sich in Abbildung 2.6.

Nun zur Erklärung, warum mit diesem Index spätestens der dritte Plattenzugriff das gesuchte Element liefert. Geht man davon aus, dass man 8kb-Blöcke verwendet und 32-bit-Zeiger zur Verfügung hat, so können in jedem Block 2048 Zeiger gespeichert werden. Wenn etwa die Hälfte innerhalb der Ebene benötigt werden, bleiben noch rund 1000 Zeiger zur Verlinkung von Blöcken auf einer tieferen Ebene. Bei einer Index Fabric mit drei Ebenen würde die Ebene 0 allein aus $1000^2$ Blöcken bestehen. Bei 1000 Knoten pro Block können mit diesem Index 1.000.000.000 Wörter indexiert werden. Die oberen beiden Ebenen haben insgesamt 1001 Blöcke mit 8kb, belegen also rund $7,8$MB Speicher. Dies ist so wenig, dass diese beiden Ebenen ohne weiteres im Hauptspeicher gehalten werden können. Weil PT-Bäume nicht die kompletten Wörter indexieren, sondern einige Buchstaben überspringen, kann es vorkommen, dass bei der Suche

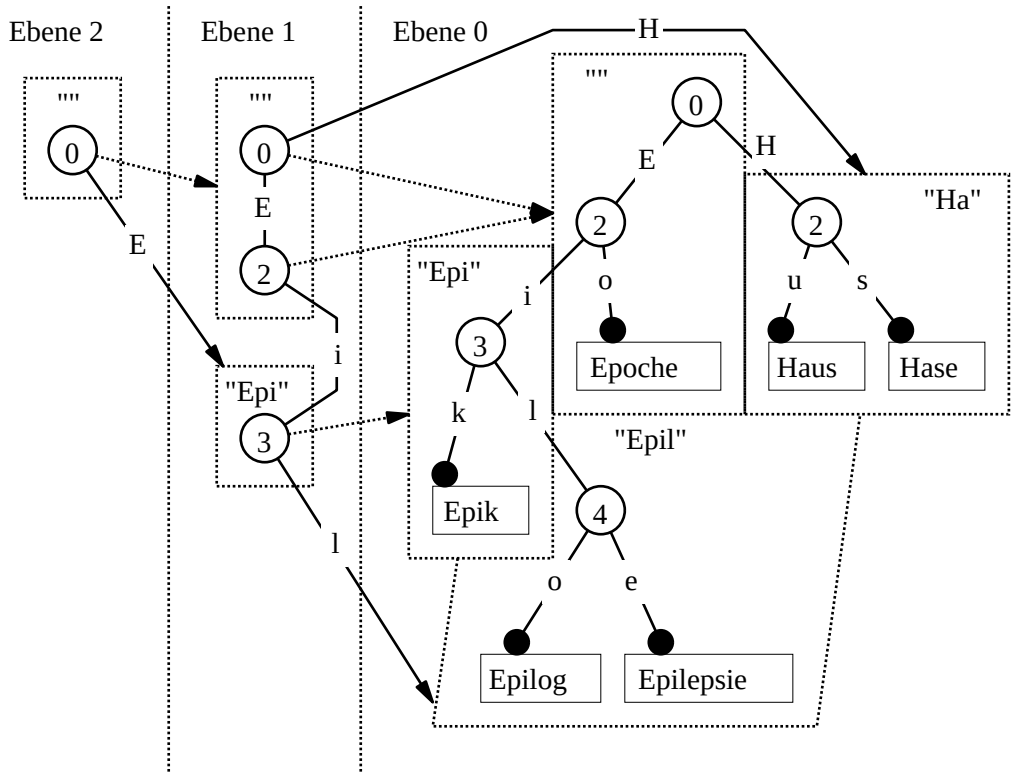



**Abbildung 2.6:** Index Fabric mit 3 Ebenen

ein falscher Pfad verfolgt wird. Da aber in jedem Block das Präfix aller folgenden Wörter abgespeichert ist, wird maximal einmal pro Ebene eine falsche Entscheidung getroffen. Spätestens der zweite Vergleich in einer Ebene schließt das Vorhandensein eines Wortes im Speicher aus oder verfolgt weiter den richtigen Weg [CS01]. Da sowohl die Ebene 0, als auch schließlich das gesuchte Wort auf der Festplatte gespeichert sind, werden also maximal drei Plattenzugriffe benötigt.

## 2.5 Bitindizes

Diese Indexierung basiert auf der Verwendung von Hashfunktionen. Mit ihnen wird jedem Wort eine Zahl zugeordnet. Die Gesamtheit aller Wörter aus allen Dokumenten wird somit auf eine Menge von Zahlen abgebildet. Nun wird ein geeignetes großes Bitfeld betrachtet. Für jede Zahl in der Menge wird das entsprechende Bit im Feld auf 1 gesetzt. Der Inhalt des Bitfeldes, als binäre Zahl betrachtet, wird dann als die Signatur dieser Zahlenmenge bezeichnet. Mit ihr kann eindeutig bestimmt werden, ob sich ein gesuchtes Wort *nicht* im Repositorium befindet. Der Umkehrschluss gilt im allgemeinen nicht, da zur Natur von Hashfunktionen gehört, dass verschiedene Eingabewerte gleiche Ausgaben produzieren können. Man kann nur abhängig von der Qualität der Hashfunktion (wenig/viele Kollisionen) bestimmen, mit welcher Wahrscheinlichkeit sich ein Wort im Repositorium befindet.

Als Beispiel sei eine Hashfunktion gegeben, die jedes Wort auf eine Zahl zwischen 0 und 7 abbildet. Der ganze Inhalt eines Hosts besteht aus dem Satz "*Die erste Generation der Funktelefone war analog.*". Die Hashfunktion wird nun auf alle die Wörter angewendet, die nicht von der Stoppworteliminierung[2] betroffen sind. Das Ergebnis könnte folgendes sein:

| | | |
|---|---|---|
| erste | = | 4 |
| Generation | = | 1 |
| Funktelefone | = | 3 |
| analog | = | 5 |

Werden nun im Bitfeld die Positionen 1, 3, 4, 5 auf 1 gesetzt, so ergibt sich eine Signatur wie in Abbildung 2.7. In das dezimale Zahlensystem umgerechnet ist die Signatur 58.

Hashfunktionen haben den Nachteil, dass die Anzahl der adressierbaren Eingabewerte begrenzt ist durch die Mächtigkeit des Definitionsbereiches der Hashfunktion. Ausnahmen sind Arten des verketteten Hashens mit Überlaufseiten oder Verfahren mit dynamischer Platzallokation [Lit78]. Eine andere Möglichkeit die Bits im Index zu adressieren sind Bloomfilter [Blo70]. Bloomfilter sind

[2]Stoppwörter sind Wörter, die mit hoher Wahrscheinlichkeit in jedem Text mehrfach vorkommen. Aus diesem Grund werden sie von der Indexierung ausgeschlossen.

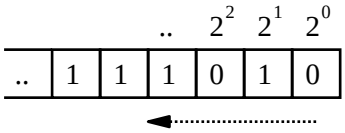



**Abbildung 2.7:** Signatur für das Beispiel: Zählweise 0, 1, 2, ..., beginnend mit dem Least-Significant-Bit.

Hashmethoden zur Reduzierung des Platzbedarfes der gehashten Eingabeobjekte. Die Basis bilden hierbei mehrere Hashfunktionen anstelle einer einzelnen. Mit der Reduzierung des Platzbedarfes geht eine Steigerung der Kollisionswahrscheinlichkeit einher.

Als Hashfunktionen werden minimale, perfekte Hashfunktionen $f$ angenommen. Eine Hashfunktion $f$ ist perfekt, wenn gilt:

$$\forall a, b \in Definitionsbereich : a \neq b \Rightarrow f(a) \neq f(b)$$

Minimal bedeutet, dass, wenn der Definitionsbereich der Funktion die Mächtigkeit $x$ hat, dann besitzt auch der Wertebereich die Mächtigkeit $x$. Umgekehrt kann gesagt werden, dass, wenn man $x$ einzeln adressierbare Bits hat, man auch mittels einer Hashfunktion maximal $x$ verschiedene Wörter adressieren kann. Wird ein Bloomfilter mit zwei minimalen, perfekten Hashfunktionen verwendet, so steigt die Anzahl der verschiedenen Wörter drastisch an. Die Anzahl der Wörter entspricht der Partialsumme der Anzahl der adressierbaren Bits. Laut Friedrich Gauß berechnet sich diese Partialsumme wie folgt: $p(x) = \frac{x(x+1)}{2}$ (Spezielle Partialsumme für Natürliche Zahlen). Da für die beiden Hashfunktionen ($f_1$ und $f_2$) gelten soll, dass

$$\forall a \in Definitionsbereich : f_1(a) \neq f_2(a)$$

müssen also mindestens zwei Bits zur Verfügung stehen. Außerdem ist es unerheblich, welche der beiden Funktionen welches Bit adressiert hat. Schränkt man dies nicht ein, kann man zwar doppelt so viele Wörter adressieren, erkauft sich dies aber ausschließlich durch zusätzliche Kollisionen. Für zwei Bits gibt es für zwei Hashfunktionen genau eine Möglichkeit ein Wort zu adressieren. Für drei Bits sind es drei, für vier sechs, und so weiter. Formal berechnet sich die Anzahl

der Wörter mit:

$$\begin{aligned} P_2(n) &= \sum_{i=1}^{n-1} i \qquad |n \geq 2 \\ &= p(n-1) \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{(n-1)n}{2} \end{aligned}$$

Als Erklärung soll hier noch erwähnt werden, dass $P_x(n)$ die Anzahl der adressierbaren Wörter berechnet, wenn $x$ Hashfunktionen verwendet werden und $n$ Speicherbits zur Verfügung stehen.

Erhöht man die Anzahl der Hasfunktionen auf drei, müssen auch mindestens drei Bits zur Verfügung stehen. Für eine sehr kleine Anzahl an Bits ist die Anzahl an Wörtern geringer als bei zwei Funktionen, aber sie steigt schneller je größer die Bitzahl ist. Mit drei Bits kann ein Wort adressiert werden, mit vier vier und so weiter. Oder formal

$$\begin{aligned} P_3(n) &= \sum_{i=1}^{n-2} \sum_{j=1}^{i} j \qquad |n \geq 3 \\ &= \sum_{i=1}^{n-2} p(i) \\ &= \sum_{i=1}^{n-2} \frac{i(i+1)}{2} \\ &= \frac{(n-2)(n-1)n}{6} \\ &= \frac{(n-2)P_2(n)}{3} \end{aligned}$$

Allgemein kann folgende rekursive Formel zur Berechnung der Anzahl adressierbarer Wörter verwendet werden:

$$P_x(n) = \frac{(n-x+1)P_{x-1}(n)}{x}$$

mit $P_0(n) = 1$. In Abbildung 2.8 sieht man, wie schnell die Anzahl der Wörter wächst. Negativ ist aber, dass mit steigender Zahl der Wörter auch die Wahrscheinlichkeit steigt, dass Wörter fälschlicherweise als indexiert betrachtet werden. Dies geschieht, wenn durch verschiedene andere Wörter unter anderem alle die Bits gesetzt werden, die laut Bloomfilter für das aktuelle Wort gesetzt sein müssten. Ist das Wort aber nicht indexiert worden, spricht man von einem *false-positive* Treffer.

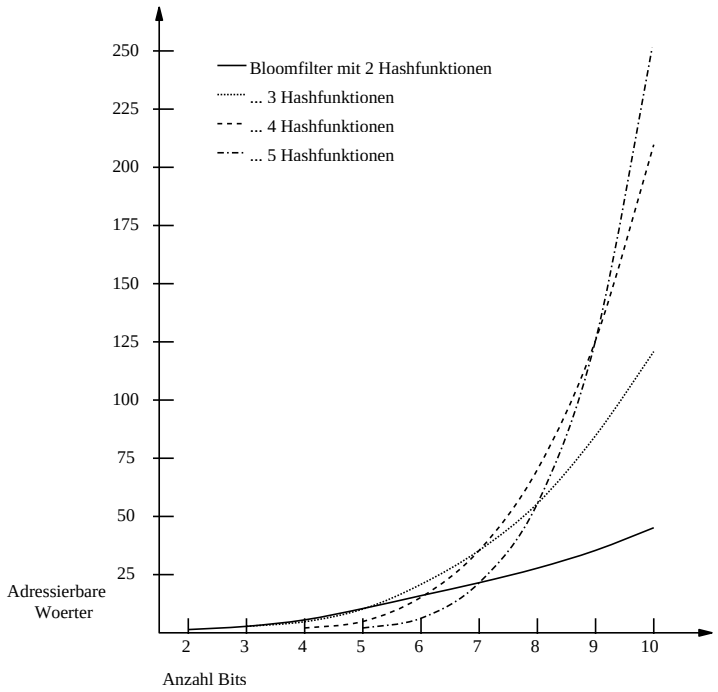



**Abbildung 2.8:** Bloomfilter Vergleich

Bitindizes bieten keine Unterstützung für die Struktur in XML-Dokumenten. Daher können die Dokumente nur Volltext indexiert werden. Es verbleibt allein die Entscheidung, ob die XML-Strukturen mit in den Volltext aufgenommen werden, oder nicht.

## 2.6 Histogramme

Histogramme werden nicht direkt zur Indexierung von Daten verwendet, vielmehr aber um Abschätzungen über das zu erwartende Ergebnis machen zu können. Gerade Optimierer in Datenbanksystemen sind auf solche Informationen angewiesen um die Laufzeit von Anfrageausführungen gering zu halten.

Für die genaue Definition eines Histogramms $H$ über ein Attribut $a$ einer Relation $R$ muss der Definitionsbereich des Attributes $D(a)$ bekannt sein. Dieser wird in $n$ Bereiche eingeteilt, welche mit $D_0(a)$ bis $D_{n-1}(a)$ bezeichnet werden. Jedem dieser Bereiche wird ein Bucket (Behälter, Balken, ...) $H_i$ zugeordnet.

Wie diese Aufteilung erfolgt, hängt von der Art des Histogramms ab. Beispiele sind Equi-Width-, Equi-Depth-, Compressed-, V-optimale- oder mehrdimensionale Histogramme. Als Grundlage für das weitere Vorgehen werden ausschließlich Equi-Width-Histogramme betrachtet.

Der Bucket $H_i$ beschreibt die relative Häufigkeit aller Elemente des Definitionsbereichsabschnittes $D_i(a)$. Das kleinste bzw. größte Element eines Bereiches ist $D_i^{\alpha}(a)$ bzw. $D_i^{\omega}(a)$. Die Breite $b$ aller Abschnitte ergibt sich aus der Anzahl der möglichen verschiedenen Werte pro Abschnitt. Da es sich bei den Histogrammen um Equi-Width-Histogramme handelt, ist die Breite aller Abschnitte gleich. Formal lassen sich die hier betrachteten Histogramme wie folgt beschreiben:

$$\forall i \; mit \; 0 \leq i < n : H_i = \frac{\gamma_{count}(\pi_a(\sigma_{D_i^{\alpha}(a) \leq a \leq D_i^{\omega}(a)}(r(R))))}{\gamma_{count}(\pi_a(r(R)))}$$

Die Semantiken von $\gamma, \pi, \sigma$ und $r$ sind der Relationenalgebra entnommen. Umgangssprachlich beschrieben beinhaltet jedes Bucket des Histogramms die Anzahl der Elemente, deren Attributwert innerhalb des zugehörigen Bereichs liegt, dividiert durch die Anzahl aller Elemente. Um dies zu verdeutlichen folgt nun ein Beispiel. Die Relation 'Person' beinhaltet das Attribut 'Alter'. Das Alter einer Person kann Werte von 1 bis 100 Jahren annehmen. Unterteilt man diesen Definitionsbereich in fünf gleich große Bereiche, so erhält man Bereiche, die jeweils zwanzig Elemente beherbergen. Für das erste Bucket wird die Anzahl der Elemente ermittelt, deren Attributwert größer gleich 1 und kleiner gleich 20 ist. Der Wert des ersten Buckets ist dann diese Anzahl dividiert durch die Anzahl aller 'Person'-en. Gleichermaßen wird nun für die weiteren vier Buckets verfahren.

Anhand von Histogrammen können Abschätzungen über numerische Daten gewonnen werden. Da aber XML-Dokumente Textdokumente sind, können mittels Histogrammen höchstens Meta-Informationen über sie abgeschätzt werden.

## 2.7 Textkompression

Mit Textkompression ist im Grunde ein Spezialfall der Dokumentreplikation gemeint. Allgemein ist die komplette Replikation aller im P2P-Netz vorhandenen Dokumente auf alle Hosts zwar theoretisch eine Lösung, praktisch jedoch ist dies unrealistisch. Um dies aber trotzdem einmal in Betracht zu ziehen muss eine Möglichkeit gefunden werden, die riesigen Datenmengen zu verkleinern. Um aber den Sinn eines P2P-Netzes nicht zu zerstören, sollen nicht komplette Dokumente sondern nur Informationen übertragen werden, die auf den Inhalt der Hosts schließen lassen. Das Szenario soll folgendes sein. Jeder Host besitzt für jeden Host einen Index. So können Anfragen auf Dokumente gezielt erfolgen, und kein Host wird unnötig belastet. Nur die Verteilung des eigenen Indexes mittels Flooding belastet das P2P-Netz anfangs stark.

Möglich wird so etwas, wenn die relative Häufigkeit aller eventuell in Dokumenten vorkommenden Wörter bekannt ist. Es muss eine Kompression gefunden werden, die nicht von lokalen Häufigkeitsverteilungen ausgeht, sondern die Verteilung auf allen Hosts berücksichtigt. Hierfür können grundsätzlich al-

le Verfahren verwendet werden, die auch lokal funktionieren, wie zum Beispiel LZ77 oder Huffmann [ZL77, Huf52]. Der Unterschied ist, dass die Häufigkeitsverteilungen nicht untersucht werden dürfen, sondern fest vorgegeben sind. Dies komprimiert die Texte zwar nicht mehr so gut, hat aber zwei Vorteile. Zum einen benötigen die komprimierten Texte kein Wörterbuch, da die Häufigkeitsverteilung allen Hosts bekannt ist. Zweitens können so auch Suchwörter komprimiert und gefunden werden.

Will man dieses Verfahren allgemeingültig verwenden, sollte man die Verteilung der Wörter anhand der zu erwartenden Sprache (Englisch, Deutsch, ...) bestimmen. Eine etwas bessere Komprimierung sollte man erhalten, wenn man alle tatsächlich vorhandenen Dokumente als Grundlage für die Bestimmung der Verteilungen nimmt.

Jeder Host komprimiert alle Dokumente seines Repositoriums und verschickt diesen kompakten Index an alle weiteren Hosts. Diese durchsuchen nun jeden ihrer empfangenen Indizes nach Vorkommen eines ebenfalls komprimierten Suchwortes. Alsdann steht exakt fest, auf welchen Hosts relevante Dokumente gespeichert sind. Hier könnten sogar Techniken aus dem Information Retrieval verwendet werden um zu bestimmen, welcher Host die interessantesten Ergebnisse liefern würde.

Wie auch immer man solch ein Verfahren umsetzen würde, so ist es doch im realen Leben völlig unbrauchbar. Denn angenommen, das P2P-Netz setzt sich aus 1.000.000 Hosts zusammen und man setzt dazu voraus, dass jeder Host es schaffen würde, sein komplettes Repositorium zu einer 1kb großen Datei zusammenzufassen. So müsste jeder einzelne Host $976,6$MB an Indexdaten speichern.

# Kapitel 3

# Routing von Anfragen

Allein ein geschickt gewählter Index, welcher angibt, welcher Host welche Daten bereitstellt, reicht nicht aus. Genauso wichtig ist das Finden einer optimalen Route vom Anfragesteller zum Zielhost. Hintergrund ist, dass es sich bei dem zugrunde liegenden Netz um ein dezentralisiertes, unstrukturiertes P2P-Netz handelt und somit kein Wissen über die Netztopologie vorhanden ist. Verwirrend scheint, warum sich die anfragestellende Anwendung selbst darum kümmern muss, wie die Anfrage zu bestimmten Host gelangt. Man sollte annehmen können, dass diese Aufgabe von der Vermittlungsschicht (ISO-OSI Referenzmodell) bzw. von der Internetschicht (TCP/IP-Referenzmodell) übernommen wird. Doch ist der Hintergrund hier ein anderer. Man kann nicht davon ausgehen, dass die im P2P-Netz gewählte Indexierungsform für Dokumente global ist. Damit ist gemeint, dass auf jedem Host ein Index liegt, anhand dessen er Dokumente auf jedem anderen Host ausfindig machen kann. Ob der eher dynamischen Netzstruktur beschränkt sich der Index nur auf die lokalen Dokumente und bezieht eventuell Nachbarn in einem kleinen Radius mit ein. Und gerade weil das Wissen so begrenzt ist, kann ein Host nicht bestimmen, welcher Host im Netzwerk die Anfrage am zufriedenstellendsten beantworten kann. Mit Routen der Anfrage ist gemeint, dass jeder Host entscheidet, welcher andere Host die Anfrage als nächster bekommt. Da es sich bei P2P-Netzen um Overlay-Netzwerke handelt, ist ein Nachbar im P2P-Netz nicht zwangsläufig ein Nachbar in dem zugrunde liegenden physischen Netz. In diesem Netz ist es dann wieder Aufgabe der oben genannten Schichten, die Anfrage an den vermeintlichen Nachbarn zu routen.

Gerade weil keinem Host die Netzwerktopologie bekannt ist und Wissen selten über die Nachbarn hinausgeht, sind bisher eingesetzte Routing-Algorithmen nur bedingt benutzbar. Andererseits existieren diese Algorithmen schon seit einigen Jahrzehnten, und man sollte sich die viele Forschung, die in sie investiert wurde, zunutze machen.

Routing-Algorithmen können in zwei Gruppen unterteilt werden. In Adaptive und Nichtadaptive Algorithmen [Tan00]. Nichtadaptive Verfahren zeichnen sich dadurch aus, dass ihre Routing-Entscheidungen nicht auf Beobachtungen

des aktuellen Verkehrs beruhen, sondern nach fest vorgegebenen Schemata verlaufen. Adaptive Algorithmen dagegen ändern abhängig von Verkehrsaufkommen oder topologischen Veränderungen ihre Routing-Entscheidungen.

Einige der folgenden Routing-Algorithmen sind Algorithmen aus Nicht-P2P-Netzen. Sie sind für den Einsatz in P2P-Netzen nicht vorgesehen und in ihnen auch nicht umsetzbar. Somit sind einige von denen hier schon für dezentralisierte P2P-Netze angepasst.

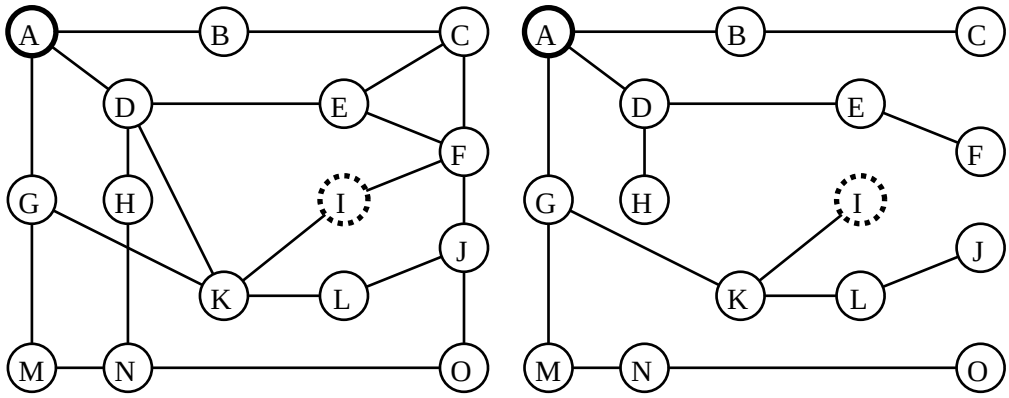
## 3.1 Flooding

Der in dezentralisierten, unstrukturierten P2P-Netzen am häufigsten eingesetze nichtadaptive Routing-Algorithmus ist das Flooding. Hierbei versendet jeder Host jedes Datenpaket über jeden Link über den er verfügt. Ausgenommen ist der Link, über den das Datenpaket angekommen ist. In der Vermittlungsschicht (ISO/OSI) oder der Internetschicht (TCP/IP) werden tatsächlich nur Datenpakete betrachtet [Tan00]. Im Fall von P2P-Netzen ist das Routen allerdings Bestandteil der Anwendungschicht, und mit Routen ist das Finden von optimalen Pfaden von der Quelle bis zum Ziel für Objekte gemeint. Objekte sind hierbei etwa Queries, Dokumente oder allgemeine Nachrichten.

Bei diesem Verfahren wird ein Objekt redundant durch das Netz "geflutet" und würde sich ohne Einschränkungen und Abbruchbedingungen unendlich oft vervielfältigen und unendlich lange leben, wobei unendlich nur die Zeit meint, bis das Netz zusammenbricht. Eine Einschränkung ist das Zuordnen eines eindeutigen Identifikators (ObjectID) zu einem Objekt. Jeder Host speichert die ObjectID eines jeden Objektes bevor er es weiter verschickt. Bekommt er ein Objekt, dessen ObjectID schon gespeichert ist, wird das Objekt verworfen. Somit wird verhindert, dass die Inkarnationen eines Objektes nicht die Anzahl aller Hosts im kompletten P2P-Netz überschreitet. Eine weitere Einschränkung ist das Einführen eines Streckenzählers[3]. Hierbei fügt der Quellhost an das Objekt einen Streckenzähler an und initialisiert ihn mit der maximalen Anzahl an Teilstrecken, die ein Objekt zurücklegen darf. Der Idealfall ist, den Streckenzähler genau auf die Länge des Pfades von der Quelle zum Ziel zu setzen. Für gewöhnlich ist diese aber nicht bekannt. Doch selbst mit diesen Einschränkungen ist die Differenz zwischen erzeugten Objekten und eigentlich benötigten Objekten sehr groß.

Die Abbildung 3.1 zeigt ein Beispiel für die unnötige Vervielfältigung eines Anfrageobjektes. Annahme ist, dass ein Objekt nur vervielfältigt werden muss, wenn ein Host das Objekt über mehrere Leitungen gleichzeitig verschicken muss. In Bild 3.1(a) ist ein Beispielnetz abgebildet. Der Host $A$ will eine Anfrage verschicken, und Host $I$ ist der einzige, der die Anfrage erfüllen wird. Mit diesem

[3]Die "Maßeinheit" des Zählers ist 'Hop'. Die Entfernung zwischen zwei direkten Nachbarn ist grundsätzlich ein Hop.




**(a)** Beispielnetz **(b)** Versandbaum

**Abbildung 3.1:** Flooding-Beispiel

Wissen würde Host $A$ die Anfrage über $D$, $K$ oder über $G$, $K$ zu $I$ schicken. Dafür ist nur ein einziges Anfrageobjekt von Nöten, und es würde das Ziel in drei Hops erreichen. Da aber kein Host über dieses Wissen verfügt, wird das Anfrageobjekt in das Netz geflutet. Mit der Einschränkung, dass kein Host ein Objekt ein zweites Mal entgegennehmen darf, entsteht von $A$ aus gesehen ein Versandbaum. Ein möglicher Versandbaum ist in Abbildung 3.1(b) zu sehen. An jeder Abzweigung wird mindestens ein neues Objekt kreiert. Letztendlich werden genau so viele Objekte kreiert, wie der Verandbaum Blätter hat. In diesem Beispiel sind dies sechs, und diese sind insgesamt vierzehn Hops weit gereist (Summe aller Entfernungen vom Blatt bis zum Knoten der Kreation). Das Ergebnis dieses Beispiels ist, dass man $83,3\%$ Anfrageobjekte und $78,6\%$ verwendete Leitungen hätte einsparen können.

Andere Ansätze beschränken im Voraus die Ergebnismenge. Als Beispiel kann hier *Interactive Deepening* erwähnt werden. Hierbei wird im Voraus die erwartete Ergebnismenge bestimmt und der Streckenzähler mit einem niedrigen Wert initialisiert. Die eigentliche Suche verläuft zunächst genauso wie oben beschrieben. Erreicht der Streckenzähler den Wert 0, wird die Anfrage für eine gewisse Zeit eingefroren und alle bisherigen Ergebnisse zum Initiator der Anfrage gesandt. Wird innerhalb dieser Zeit kein Befehl zur Weiterverarbeitung empfangen, wird die Anfrage verworfen. Entspricht das bisher erreichte Ergebnis nicht mindestens der erwarteten Erbebnismenge, so wird eine *RESEND MESSAGE* mit einem neuen Wert für den Streckenzähler geflutet. Alle eingefrorenen Anfragen werden nun wieder normal abgearbeitet bis der Streckenzähler abermals 0 erreicht. Dies wird solange wiederholt, bis das Ergebnis befriedigend ist.

Ein anderer Ansatz ist, dass sich jeder Host speichert, wie viele Ergebnisse welcher Nachbar in der Vergangenheit geliefert hat. Auch hier werden im

Voraus die Ergebnismenge und der Streckenzähler festgelegt. Anders als bei dem Interactive Deepening wird hierbei aber nicht in die Breite gesucht und der Streckenzähler bei Bedarf erhöht, sondern es wird in die Tiefe gesucht. Zunächst wird die Anfrage an den Nachbarn geleitet, der in der Vergangenheit die meisten Ergebnisse zurückgegeben hat. Dieser geht genauso vor, solange bis der Streckenzähler genullt ist. Reicht das bisher erbrachte Ergebnis nicht aus, wird nun die Anfrage an den Nachbarn geschickt, der in der Vergangenheit die zweitmeisten Ergebnisse geliefert hat.

Auch wenn beide Ansätze die Netzlast senken, so schränken sie doch die Ergebnismenge ein. Und gerade der zweite Ansatz hat das Problem, dass seltene Dokumente, oder Dokumente auf Hosts, die wenig Dokumente anbieten, eventuell gar nicht gefunden werden.

## 3.2 Distance-Vector-Routing

Ein bis 1979 sehr häufig eingesetzter Routing-Algorithmus war das Distance-Vector-Routing (DVR) [Tan00]. Dieser ist ein adaptiver Algorithmus und zeichnet sich dadurch aus, dass seine Routing-Entscheidungen von dem Verhalten des Netzes abhängig sind. Da dieser Algorithmus nur bis 1979 eingesetzt wurde, ist davon auszugehen, dass er mittlerweile überholt und durch bessere Algorithmen ersetzt wurde. Als Beispiel sei hier das Link-State-Routing erwähnt. Doch da die moderneren Algorithmen der gleichen Klasse von Routing-Verfahren angehören und im Endeffekt von der Idee her ähnlich sind, soll das DVR als Vertreter für klassische, adaptive Routing-Algorithmen dienen.

Für das DVR benötigt jeder Host eine Tabelle, in der alle bekannten Entfernungen zu anderen Hosts gespeichert sind. Was genau diese Entfernungen beschreiben sei hier nicht beschrieben. Dies kann die Anzahl der Hops oder auch die Sendezeit zum Ziel sein. Die Tabelle wird je nach Erfahrungen aktualisiert, und die Routing-Entscheidungen können sich somit ändern. Jede Zeile der Tabelle besteht aus drei Spalten. Der Host-, der Nachbar- und der Entfernungs- oder Verzögerungsspalte. Für jede Anfrage wird in der Tabelle nach dem Adressaten gesucht, und diese wird dann über den in der gleichen Zeile angegebenen Nachbarn weitergeleitet. Die Tabelle wird kontinuierlich parallel zur normalen Arbeit des Hosts aktualisiert. Dieser Prozess ist in Algorithmus 1 dargestellt.

Das DVR hat den Vorteil, dass Informationen über neue Hosts sich schnell über das ganze Netz verbreiten. Nachteil ist aber, dass das Verschwinden eines Hosts erst sehr spät bemerkt wird. Beides liegt in dem Algorithmus 1 verborgen. Betritt ein Host das Netz, so kennen ihn nach dem ersten Durchgang nur seine direkten Nachbarn. Wird das Wissen darüber, dass es den Host gibt, als Wissenshorizont bezeichnet, so hat er nach dem ersten Durchgang einen Radius von einem Hop. Mit jedem Durchgang erhöht sich der Radius um einen weiteren Hop. Tritt ein Host aber unerwartet aus dem Netz aus, so ist dies für andere

1: Der Host ($H$) trägt sich selbst in seine leere Tabelle ein. Die Spalte des Nachbarn bleibt leer und die Verzögerung wird mit 0 angegeben.
2: **while** der Tabellenaufbauprozess nicht beendet wird **do**
3: $H$ ermittelt die mittlere Versandzeit zu jedem Nachbarn. $((Ankunftszeitpunkt(pong) - Versandzeitpunkt(ping))/2)$.
4: Versenden der aktuellen Tabelle an alle Nachbarn.
5: Empfangen der Tabellen von allen Nachbarn.
6: Anlegen einer neuen Tabelle.
7: **for** jeden Host in jeder empfangenen Tabelle **do**
8: **if** der Host ist nicht $H$ selbst **then**
9: **if** der Host ist in der neuen Tabelle noch nicht enthalten **then**
10: Eintragen des Hosts in die neue Tabelle. Als Nachbar wird der Besitzer der aktuell bearbeiteten Tabelle eingetragen. Die Verzögerung ist jene aus der bearbeiteten Tabelle addiert mit der mittleren Versandzeit zum Nachbarn.
11: **else**
12: Berechnung der Verzögerung mit dem Wert aus der bearbeiteten Tabelle addiert mit der mittleren Versandzeit zum Besitzer der Tabelle.
13: Vergleiche berechneten Wert mit dem in der neuen Tabelle.
14: **if** der berechnete Wert ist kleiner **then**
15: Ersetzen des Wertes in der neuen Tabelle mit dem Errechneten.
16: Ersetzen des Nachbarn mit dem Besitzer der aktuell bearbeiteten Tabelle.
17: **end if**
18: **end if**
19: **end if**
20: **end for**
21: $H$ trägt sich selbst in die Tabelle ein (Siehe Schritt 1).
22: Ersetzen der alten mit der neuen Tabelle.
23: Der Host wartet eine gewisse Zeit.
24: **end while**

**Algorithmus 1:** Distance-Vector-Routing

Hosts nicht einfach festzustellen. Es tritt ein Phänomen auf, welches Count-to-Infinity genannt wird [Tan00]. Ein Beispiel ist in Abbildung 3.2 beschrieben.

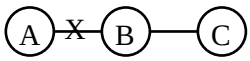



**Abbildung 3.2:** Count-to-Infinity-Problem: Verlässt Host $A$ unerwartet das Netz, so bemerkt dies Host $B$ bei der nächsten Tabellenaktualisierung, da er von $A$ keine erhält. Dies ist scheinbar nicht weiter schlimm, denn Nachbarhost $C$ teilt eine aktive Verbindung zu $A$ mit. $B$ weiß aber nicht, dass diese Route über sich selber geht. Im nächsten Schritt bemerkt $C$, dass seine Route zu $A$ über $B$ länger dauert als bisher und ändert dies in seiner Tabelle. Einen Schritt später passt nun $B$ wiederum seinen Eintrag an, u. s. w.

Dieser Algorithmus ist in der Lage Anfragen schnell zum Adressaten zu befördern. Grundvoraussetzung ist, dass der Sender weiß, wohin eine Anfrage oder ein Objekt gesendet werden soll. Er bietet somit keinerlei Unterstützung für Anfragen in dezentralisierten P2P-Netzen. Sie müssen nach wie vor per flooding verteilt werden. Es wird einzig die Netzlast bei dem Austausch der Objekte minimiert. Um die Auswirkungen des Count-to-Infinity-Problems gering zu halten, können zwei Hilfen eingeführt werden. Zum einen die Definition einer maximalen Verzögerung. Ist dieser Wert überschritten, gilt ein Host als nicht vorhanden. Zum anderen das Einführen von eindeutigen Hostidentifikatoren. Verschwindet ein Host an einer Stelle des Netzes und taucht kurze Zeit später an anderer Stelle auf, so können von dieser neuen Position ausgehend alle Einträge für diesen Host berichtigt werden.

## 3.3 Inhaltsbasiertes Anfrage-Routing in Baumnetzen

Grundlage dieses Algorithmus' ist ein baumstrukturiertes P2P-Netz. Er beschreibt das approximative Indexieren von Knoten- und Teilbauminhalten sowie das Routen von Queries basierend auf diesen Indizes [Pri01]. Ein Host im Netz übernimmt die Rolle des Wurzelknotens, jeder weitere Host ist Kind eines anderen Hosts und hat wiederum ein oder mehrere Kindknoten. Für den Fall, dass ein Host ein Blatt im Baum ist, hat er keine Kindknoten. Abbildung 3.3 zeigt ein Beispiel eines solchen Netzes. Dieser Algorithmus routet abhängig von dem Repositorium der Hosts und dem Inhalt der Anfrage diese effizient durch den Netzbaum. Hierbei bedeutet Effizienz, dass eine Anfrage über so wenig Hosts wie möglich und so viele wie nötig geroutet wird. Um dies zu erreichen, ist der

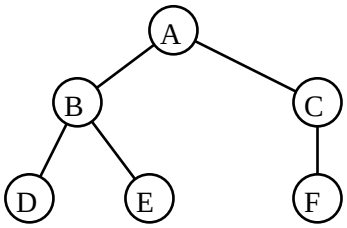
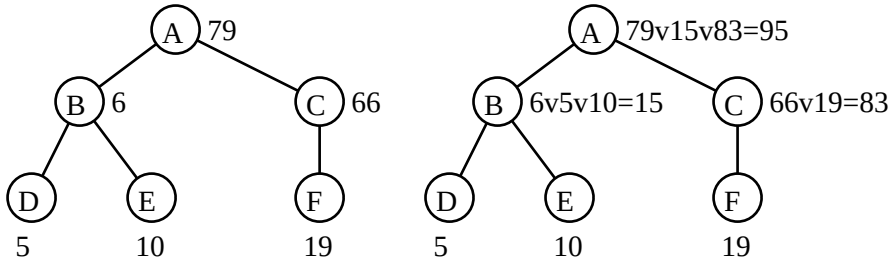



**Abbildung 3.3:** Beispiel eines baumstrukturierten P2P-Netzes




**(a)** Startzustand **(b)** Endzustand

**Abbildung 3.4:** Signaturpropagierung

Algorithmus eng an eine spezielle Art und Weise der Indexierung des Inhalts der Hosts geknüpft.

Diese Art der Indexierung wurde bereits in Sektion 2.5 vorgestellt. Zur Erinnerung sei gesagt, dass jeder Host, abhängig von dem Inhalt seines Repositoriums, eine mit hoher Wahrscheinlichkeit eindeutige Signatur besitzt. Man kann nicht sagen, dass sie eindeutig ist, da durchaus die Wahrscheinlichkeit besteht, dass zwei Hosts exakt den gleichen Inhalt haben.

Verfügt ein Host über Kindknoten, verlangt er nun die Signaturen von diesen und verknüpft sie und seine eigene mittels der logischen Funktion *OR* miteinander. Nach dieser Operation werden die einzelnen Operanden abgespeichert, denn sie abstrahieren von dem Inhalt, welcher in den einzelnen Teilbäumen zu finden ist. Wird ein Host von seinem Vaterknoten aufgefordert, so sendet er nicht seine Signatur, sondern die verknüpfte. In Abbildung 3.4 ist dieser Vorgang dargestellt.

Das eigentliche Routing verläuft nun folgendermaßen. Der anfragestellende Host generiert für die Anfrage eine Signatur. Wichtig ist hierbei, dass exakt die selbe Hashfunktion verwendet wird wie zur Erstellung der Hostsignaturen. Diese und die eigentliche Anfrage werden an den Vaterknoten weitergeleitet. Er geht nun wie in Algorithmus 2 beschrieben vor.

```
1: if Host ist nicht Wurzel then
2:     Weitergabe der Anfrage und der Anfragesignatur an den Vater.
3: end if
4: for jeden Kindknoten außer dem, welcher die Anfrage geschickt hat do
5:     if Signatur_Kind ∧ Signatur_Anfrage = Signatur_Anfrage then
6:         Weitergabe der Anfrage und der Anfragesignatur an den Kindknoten
7:     end if
8: end for
9: if Signatur_Selbst ∧ Signatur_Anfrage = Signatur_Anfrage then
10:    Verarbeite die Anfrage.
11: end if
```

**Algorithmus 2:** Berechnung der Routing-Entscheidung

Das Verfahren birgt zwei Arten des falschen Routings in sich. Die erste ist in dem Wesen von Hashfunktionen begründet. Es kann vorkommen, dass eine Hashfunktion für verschiedene Eingabewerte den gleichen Ausgabewert berechnet. So kann fälschlicherweise angenommen werden, dass ein Wort vorhanden ist. Dieses Problem lässt sich durch einen genügend großen Wertebereich minimieren oder eliminieren. Grundsätzlich gilt, je weniger Kollisionen bei der Berechnung, desto weniger falsche Routing-Entscheidungen dieser Art. Eine weitere Möglichkeit dieses Problem zu reduzieren ist der Einsatz von Bloomfiltern [Blo70]. Hierbei wird jedes Wort mittels mehrerer verschiedener Hashfunktionen indexiert. Zwei Wörter, für die die erste Hashfunktion eine Kollision berechnet, sind bei den anderen Hashfunktionen kollisionsfrei.

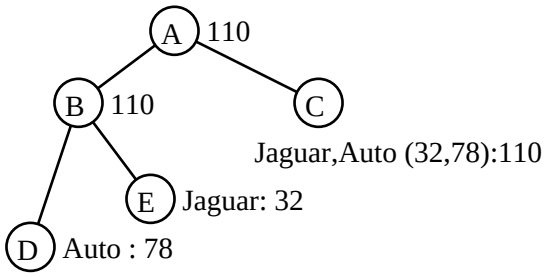



**Abbildung 3.5:** Beispiel für falsches Routing durch Suchwortaggregation: Wird nach "Jaguar2 und "Auto" gesucht, wird die Anfrage von $A$ auch nach $B$ geleitet, da $A$ nicht sehen kann, dass sich der Index aus verschiedenen Indizes zusammensetzt.

Die zweite Art des falschen Routings ist auf das Aggregieren von Suchwörtern zurückzuführen. Ein Beispiel findet sich in Abbildung 3.5. Die Hashfunktion hat für das Wort 'Jaguar' 25 berechnet und für 'Auto' 78. Im rechten Teilbaum

befindet sich ein Host, der Dokumente über die Automarke Jaguar bereit hält. Die Anfrage wird korrekt dorthin geroutet. Im linken Teilbaum befindet sich ein Host, welcher Dokumente über Tiere, und ein Host, der Dokumente über Schneeketten hält. Diese beiden Informationen werden bei dem ersten gemeinsamen Vorfahren zusammengefasst und als eine Information weiter nach oben gereicht. Dort ist dann nicht mehr entscheidbar, ob die Information von nur einem oder mehreren Hosts kam. Die Lösung liegt in einer Hashfunktion, welche eindeutige Informationen über die Hosts beachtet. Jeder Host besitzt einen eindeutigen Knotenidentifikator (KID). Dieser ist Bestandteil der Hashfunktion und einziger Unterschied in der sonst global gleichen Funktion. Resultat ist, dass auf jedem Host dasselbe Wort einem anderen Hashwert zugeordnet wird. Der Bitindex wird um den KID erweitert. Beide werden zusammen zum Vaterknoten gereicht, und dieser kann nun für jedes Suchwort, für jeden KID bestimmen, ob das Suchwort im entsprechenden Teilbaum vorhanden ist. Genau wie bei den Bloomfiltern gilt hierbei, dass ein nicht gefundenes Wort garantiert nicht vorhanden ist, aber das Gegenteil gilt nicht.

Das ganze Verfahren ist gewissermaßen statisch. Die Signaturen werden einmalig generiert und verbreitet. Von daher ist der Algorithmus ungeeignet für sich relativ oft ändernden Inhalt. In diesem Fall müsste die Signatur neu erstellt und bis zum Wurzelknoten propagiert werden. Viel ungünstiger stellt sich die Problematik des unkontrollierten Ein- und Austritts von Knoten im Netz dar. Jeder Knoten kennt maximal seinen Vater und seine Kinder. Selbst weitere Kinder des Vaters sind unbekannt. Verlässt ein Knoten das Netz, entstehen viele kleinere Netze, wobei jedes Kind zum Wurzelknoten wird. Da das P2P-Netz unstrukturiert ist, gibt es kein Wissen über die Topologie. Betritt ein Host das Netz, hat er keine Ahnung von einem hierarchischen Indexbaum. Auch kennen sie nur die Hosts in der unmittelbaren Umgebung, was es sehr schwer macht, sich auf einen Wurzelknoten zu einigen.

## 3.4 Histogramm-Routing

[PKP04]. Für diese Art des Routings benötigt jeder Host zwei Arten von Indizes, einen lokalen Index und für jeden seiner Nachbarn einen Routing-Index. Grundlage dieser Indizes sind Histogramme. Sie dienen auch dazu, Hosts mit ähnlichem Inhalt zu Gruppen zusammenzufassen (clustering). Im Gegensatz zu herkömmlichen adaptiven Routing-Algorithmen passt sich dieser nicht an das Netz an, sondern er passt das Netz an sich an. Dies geschieht, indem Hosts, welche zwar im Netz vorhanden aber nicht bekannt sind, zur Nachbarschaft hinzugefügt werden, beziehungsweise Nachbarn aus eben dieser entfernt werden. Da ein Nachbar ein Host mit direkter Verbindung ist, wird einfach eine solche Verbindung erzeugt oder gelöscht.

Jeder Host $n$ verfügt über ein Histogramm $LI(n)$ (vergleiche Sektion 2.6)

über sein Repositorium. Desweiteren erstellt er für jeden seiner Nachbarn $N_i$ ($1 \leq i \leq AnzahlNachbarn$) einen Routing-Index $RI(n, N_i)$. In $RI(n, N_i)$ werden alle Repositoriumsinformationen beachtet, die auf Hosts gespeichert sind, welche sich in Reichweite von $r$ Hops über $N_i$ befinden. Anhand Abbildung

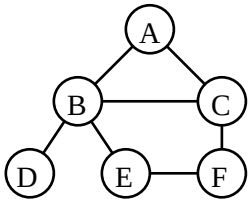



**Abbildung 3.6:** Reichweitenbeispiel

3.6 kann die Reichweite erklärt werden. Betrachtet wird der Host $A$, und die Reichweite wird auf 2 festgelegt. Da Host $A$ zwei direkte Nachbarn hat, legt er auch zwei Routing-Indizes an. Der erste Nachbar ist Host $B$. Im ersten Routing-Index werden nun die Informationen aller der Hosts verarbeitet, die sich in zwei Hops über den Host $B$ erreichen lassen. Dies sind $B$, $C$, $D$ und $E$. Über Host $C$ lassen sich innerhalb zwei Hops nur $B$, $C$ und $F$ erreichen. Die Menge der so erreichbaren Nachbarn wird als Horizont $H_{N_i,r}$ über $N_i$ bezeichnet.

Ist der Horizont über einen Nachbarn bekannt und liegen für alle Hosts $h$ innerhalb des Horizonts die Histogramme $LI(h)$ vor, so kann der Routing-Index für den Nachbarn wie in Definition 3.4.1 berechnet werden.

**Definition 3.4.1** *Haben alle lokalen Histogramme im Horizont die gleiche Anzahl Buckets b, und $H_j(n)$ (mit $H \in \{LI, RI\}$) steht für das $j$-te Bucket von $H$, dann definiert*

$$\forall j \; mit \; 0 \leq j < b : RI_j(n, N_i) = \frac{\sum_{h \in H_{N_i,r}} LI_j(h)}{|H_{N_i,r}|}$$

*einen Routing-Index für n für den Nachbarn $N_i$. Da alle Histogramme die gleiche Größe haben, gilt auch*

$$\forall h \in H_{N_i,r} : |RI(n, N_i)| = |LI(h)|$$

Um spätere Anfragen effizienter abarbeiten zu können, werden die Hosts des Netzes nach ähnlichem Inhalt gruppiert. Da der lokale Index den Inhalt eines Hosts wiederspiegelt, wird diese Gruppierung anhand dieser Indizes vollzogen. Hierfür werden zwei Metriken eingeführt. Die $L_1$-Distance und die Edit-Distance [PKP04]. Die $L_1$-Distance zwischen zwei Histogrammen beschreibt die Summe der Unterschiede zwischen den einzelnen Buckets. Formal ist sie in Definition 3.4.2 beschrieben.

**Definition 3.4.2** *Gegeben sind zwei Histogramme $H(n_1)$ und $H(n_2)$ (mit $H \in \{LI, RI\}$) mit jeweils $b$ Buckets. Ihre $L_1$-Distance $d_{L_1}(H(n_1), H(n_2))$ ist wie folgt definiert:*

$$d_{L_1}(H(n_1), H(n_2)) = \sum_{i=0}^{b-1}(|H_i(n_1) - H_i(n_2)|)$$

*Und $|x|$ bezeichnet den absoluten Betrag von $x$.*

Die $L_1$-Distance gibt zwar an, wie stark sich zwei Histogramme unterscheiden, nicht aber wo. Als einfaches Beispiel werden drei Histogramme mit jeweils vier Buckets betrachtet. Alle Buckets haben den Wert 0, außer das erste Bucket des ersten Histogramms, das zweite Bucket des zweiten Histogramms und das vierte Bucket des dritten Histogramms. Diese Buckets haben den Wert 1. Es kann nachgerechnet werden, dass die $L_1$-Distance zwischen allen Histogrammen gleich ist. Intuitiv ist aber der Abstand zwischen dem ersten und dem zweiten Histogramm kleiner als zwischen dem ersten und dem dritten. Die Edit-Distance zwischen zwei Histogrammen beschreibt nun, wieviele Schritte notwendig sind, um das eine in das andere zu überführen. Dies ist formal in Definition 3.4.3 beschrieben.

**Definition 3.4.3** *Gegeben sind zwei Histogramme $H(n_1)$ und $H(n_2)$ (mit $H \in \{LI, RI\}$) mit jeweils $b$ Buckets. Ihre Edit-Distance $d_e(H(n_1), H(n_2))$ ist wie folgt definiert:*

$$d_e(H(n_1), H(n_2)) = \sum_{i=0}^{b-1}\left(\left|\sum_{j=0}^{i}(H_i(n_1) - H_i(n_2))\right|\right)$$

*Und $|x|$ bezeichnet den absoluten Betrag von $x$.*

Wird nun mit jeder Antwort auf eine Anfrage auch das $LI(a)$ Histogramm des antwortenden Hosts $a$ mitgesandt, so kann der anfragestellende Host $n$ anhand der beiden Distanzen feststellen, ob $a$ in seinem Repositorium einen ähnlichen Inhalt wie $n$ hat. Ist dies der Fall wird eine direkte Verbindung zwischen den beiden Hosts aufgebaut (und eventuell eine Verbindung zu einem bisherigen Nachbarn gelöst). Auf diese Art bilden sich mit der Zeit Gruppen von Hosts mit ähnlichem Inhalt. Um zu vermeiden, dass sich viele isolierte Inseln bilden, muss jeder Host mindestens eine Verbindung zu einem Host behalten, der eben keinen ähnlichen Inhalt besitzt. Nun kann jeder Host entscheiden, ob eine Anfrage von seinen Gruppenmitgliedern beantwortet werden kann. Wenn ja, dann kann die Anfrage innerhalb der Gruppe bleiben und wird wahrscheinlich befriedigend beantwortet werden. Wenn nicht, so wird sie über eine der gruppenexternen Verbindungen verschickt.

# Kapitel 4

# Ansatz zur Umsetzung eines Prototypen

In diesem Kapitel soll ein Prototyp für einen XML-Speicher entwickelt werden, der als Aufsatz (Erweiterung) eines bestehenden fungiert. Er soll als Vermittlungsebene in einem dezentralisierten P2P-Netz dienen. Die Aufgaben des Prototypen sind die Indexierung von lokalen Daten, die Propagierung des Indexes im Netz sowie das geschickte Routing von Anfragen zu den Informationsträgern. Die Entwicklung wird in drei Schritten erfolgen. Im ersten Schritt wird die Verwendbarkeit der verschiedenen Formen der Indexierung für den Prototypen geprüft. Danach wird das Routing der Anfragen und abschließend die Anfrageabarbeitung diskutiert.

## 4.1 Indexierung

Hier nun werden die verschiedenen Ansätze aus Kapitel 2 betrachtet. Da diese nicht zwangsläufig für P2P-Netze konstruiert wurden, müssen sie eventuell noch angepasst werden.

Zunächst muss die Frage geklärt werden, mit welchem Schema-Szenario der Prototyp konfrontiert sein wird. Ein global-globales Schema ist der Best-Case und würde die Entwicklung vereinfachen. Doch wäre der Prototyp dann nur unter genau dieser Konfiguration operabel. Wird aber von Anfang an davon ausgegangen, dass es kein Schema gibt (Worst-Case), sollte der Prototyp sowohl mit als auch ohne Schema arbeiten, wobei er dann zwar nicht das Schema zur Hilfe nimmt, es aber ignoriert.

Zur Frage, welche Indexierungsform zum Einsatz kommen wird, werden für alle Vorgehensweisen die Vor- und Nachteile erläutert. Der erste mögliche Index sind Data-Guides (DG). Data-Guides sind ideal, um negativ auf Anfragen antworten zu können, ohne das Repositorium durchsucht zu haben. Sie stellen in gewisser Hinsicht eine Disjunktion der Struktur aller Dokumente eines Hosts dar. Und mit ihnen kann für Anfragen der Form

```
<xsl:value-of select="/CustomerOrder/CustomerInformation/
 Address[@type='shipping']/City">
```

sehr schnell entschieden werden, ob sich im Repositorium überhaupt Dokumente mit dem Pfad `/CustomerOrder/CustomerInformation/Address/City` befinden. Etwas, was DG aber nicht unterstützen, sind Volltextanfragen. Etwa die Anfrage

```
"Puppe AND Schmetterling"
```

Somit kann ein fremder Host keine Vorentscheidung treffen, an wen die Anfrage weitergeleitet werden sollte. Die Indexierung soll aber genau diese Entscheidung möglich machen. Es darf nicht vergessen werden, dass DG in Lore zur Speicherung von Metadaten und nicht zur eigentlichen Indexierung verwendet werden. In Lore stecken viele Jahre Forschung, und würde es einen effizienten Weg geben, mit DG Daten zu indexieren, so wäre dies in dem System umgesetzt.

Für die weiteren Beschreibungen wird der zu erwartende Speicherverbrauch der Indizes berechnet. Als grundlegende Annahme soll gelten, dass es 300.000 zu indexierende alphanumerische Wörter gibt. Diese Anzahl beruht darauf, dass der Prototyp grundsätzlich in der Lage sein soll, jedes englische Dokument zu indexieren, und die Anzahl aller bekannten englischen Wörter wird allgemein mit 300.000 angegeben.

Standard-Tries sind alleine wegen ihrer Größe für die Aufgabe nicht geeignet. Es darf nicht vergessen werden, dass es sich zwar um die Indexierung lokaler Daten handelt, doch sollen von diesem Index später hauptsächlich fremde Hosts profitieren. Hier sind PATRICIA-Tries eher geeignet. Sie zeichnen sich durch eine kompakte Speicherung aus. Noch kompakter wird der Index, wenn man die eigentlichen Datensätze weglässt. Es ist dann zwar nicht mehr überprüfbar, ob ein gefundenes Wort auch tatsächlich vorhanden ist, der Platzbedarf für den Index sinkt jedoch drastisch. Werden alle Wörter indexiert, und im Trie ist ein Zeiger 32 Bit groß, dann werden insgesamt zwischen $36^3$ und $36^4$ Zeiger gebraucht. Das macht eine Indexgröße von 182kb bis $7,1$MB. Aber selbst die untere Grenze ist noch zu groß, als dass solche Indizes vielfach auch in Nicht-Breitbandnetzen ausgetauscht werden.

Der Platzbedarf für Bitindizes steht von Anfang an fest und ändert sich auch nicht mehr. Alleine diese Eigenschaft macht Bitindizes sehr verlockend. Auch der zu erwartende Platzbedarf ist niedrig. Wird als Hashfunktion eine minimale, perfekte Hashfunktion verwendet, sind Kollisionen ausgeschlossen und die Anzahl benötigter Buckets ist gleich der Mächtigkeit der Eingabemenge (alle zu erwartenden Wörter). Um ein Wort im Bitindex abzuspeichern, wird genau ein Bit gebraucht. Für die angenommene Anzahl aller Wörter wird also ein Speicherplatz von $36,6$kb benötigt. Alleine das Zulassen auf durchschnittlich vier Kollisionen pro Bucket ergibt einen Indexspeicherbedarf von unter 10kb.

Noch mehr Platz lässt sich durch die Verwendung von Histogrammen einsparen. Wird der Wertebereich aller Wörter in 10.000 Bereiche eingeteilt, so bedarf

es eines Arrays aus ebensovielen Zahlen. Zur Abspeicherung dieser Zahlen reicht es aus, einen 1 Byte Datentyp zu verwenden. Jede Zahl liegt zwischen 0 und 30, und selbst ein vorzeichenbehafteter Datentyp bräuchte hierfür nur 6 Bits. Somit ist in diesem Beispiel der Speicherplatz auf nicht einmal 10kb beschränkt.

Die Entscheidung ist letztendlich auf zwei Verfahren gefallen. Zum einen auf einen Histogrammindex, da er sehr kompakt ist, und abhängig von ihm sehr gut Routing-Entscheidungen getroffen werden können. Zum anderen auf einen Bitindex, da für den Histogrammindex ein wichtiger Teil Wissen fehlt. Theoretisch sind zwar alle Wörter einer Sprache bekannt, aber schon ein Rechtschreibfehler oder ein neues Fachwort sollte dem Histogrammindex Probleme bereiten. Dieses Problem hat der Bitindex nicht, und um auf die Vorteile des Histogrammindexes nicht verzichten zu müssen, wird hier ein zweistufiger Index verwendet. Die erste Stufe ist ein Bitindex, welcher ob der Hashfunktion auch ausgedachte, alphanumerische Wörter verarbeiten kann. Die zweite Stufe ist ein Histogrammindex, da nun die Eingabemenge genau abgegrenzt ist.

### 4.1.1 Bitindex

Als erstes ist die Größe des Indexes zu betrachten. Ein zu kleiner Bitindex ist später nicht aussagekräftig, da später sehr viele Kollisionen berechnet werden. Außerdem kann der Bitindex relativ schnell zu einem reinen 1-er Feld entarten. Ein zu großer Bitindex nimmt dagegen unnötigen Speicherplatz ein. Ein Gewicht für die Größe des Indexes ist die Anzahl der zu indexierenden Wörter. Wird hier eine komplette natürliche Sprache verwenden, so müssen zum Beispiel im Englischen bis zu 300.000 und im Deutschen sogar bis zu 500.000 Wörter indexiert werden können. Für konkrete Anwendungsbereiche kann der Wortschatz im Voraus stark eingeschränkt werden. Beinhalten zum Beispiel alle Dokumente im Repositorium Wissen über das Falten von Papierbooten, dann wird die Mächtigkeit des Wortschatzes nur bei einigen wenigen Tausend liegen.

Ein weiteres Kriterium ist die Güte der Hashfunktion. Güte meint in diesem Zusammenhang, wie gleichverteilt die Hashwerte berechnet werden und wie viele/wenig Kollisionen auftreten. Die idealen Hashfunktionen sind in der Klasse der minimalen, perfekten Hashfunktionen zu suchen [FCDH91]. Aus Sektion 2.5 ist bekannt, dass eine Hashfunktion $H_p(x)$ perfekt ist, wenn gilt: $\forall a, b \in Definitionsbereich : a \neq b \Rightarrow H_p(a) \neq H_p(b)$. Und minimal bedeutet, dass, wenn der Definitionsbereich der Funktion die Mächtigkeit $x$ hat, dann hat auch der Wertebereich die Mächtigkeit $x$. Perfekt setzt voraus, dass die Mächtigkeit des Wertebereiches größer gleich der Mächtigkeit des Definitionsbereiches ist. Sowohl perfekte als auch minimale Funktionen sind aber im konkreten Anwendungsfall nicht verwendbar. Minimalität kann nicht gewährleistet werden, da der komplette Wortschatz nicht bekannt ist. Es kann nur als gegeben angenommen werden, dass sich der Inhalt der Dokumente im Bereich Informa-

tik ansiedeln lässt und in englischer Sprache verfasst ist[4]. Und da Minimalität nicht gegeben werden kann, ist es schwer eine perfekte Funktion zu finden. Um sicher zu gehen, dass keine Kollisionen auftreten werden, sollte der Wertebereich übertrieben groß gewählt werden. Doch dies ist zum einen wahrscheinlich eine unnötige Verschwendung von Speicherplatz und zum anderen eventuell noch gar nicht groß genug.

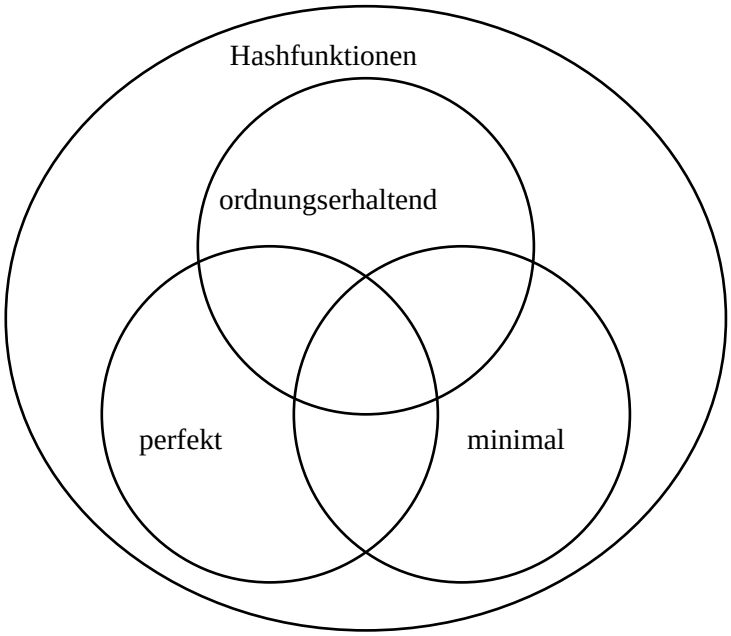



**Abbildung 4.1:** Aufteilung der Hashfunktionen in Klassen

Aus der Datenbankliteratur ist bekannt, dass sich Indizes auf Basis von Hashfunktionen nicht gut für Bereichsanfragen eignen [HS99b]. Dies liegt an der Eigenschaft, dass eine gute Hashfunktion die Eingabewerte weit verstreut auf den Wertebereich verteilt um dicht besetzte Bereiche zu vermeiden. Die Unterklasse der ordnungserhaltenden Hashfunktionen (OEH) ($H_o(x)$) besitzt als zusätzliche Eigenschaft:

$$\forall a, b \in Definitionsbereich : a < b \Rightarrow H_o(a) \leq H_o(b)$$

In der Regel widerspricht die Erhaltung der Ordnung der Qualität der Streuung. Ist aber der Definitionsbereich hinreichend bekannt, kann eine OEH konstruiert werden, welche den Verlust der guten Streuung dezimiert. Hierfür sind gewisse

[4]Für die Tests mit dem Prototypen werden ausschließlich Dokumente der INEX-Kollektion verwendet [IEE04]. Diese sind sowohl in englischer Sprache verfasst als auch aus dem Bereich der Informatik.

Metainformationen über die Dokumente notwendig. Dazu gehören Informationen darüber, in welcher Sprache sie verfasst sind und welche Wörter mit welcher Häufigkeit auftreten.

Mit

$$H(key) = \left\lfloor B \sum_{i=1}^{x} \left( \frac{p_i - 1}{2^{5i}} \right) \right\rfloor$$

wird eine triviale Funktion angegeben, um Wörter (=key) einer Zahl zuzuordnen [Rob86]. Hierbei wird jedem Buchstaben des Alphabetes ein Zahlenwert $p$ zugeordnet, zum Beispiel die Position im Alphabet. Als Alphabet wird in dieser Arbeit die geordnete Menge der Buchstaben von $a$ bis $z$ und der Zahlen von 0 bis 9 verwendet. Des weiteren wird das Eingabewort in seine $x$ Buchstaben zerlegt, und $p_i$ gibt den zugeordneten Zahlenwert des $i$-ten Buchstaben des Wortes an. Die Mächtigkeit des Wertebereiches wird mit $B$ angegeben. Diese Funktion ist weder minimal noch perfekt noch ordnungserhaltend. Auch ist zu beobachten, dass durch die Division durch $2^{5i}$ die Bedeutung der Buchstaben abnimmt je weiter hinten sie im Wort stehen. Genau genommen ändert sich das Ergebnis ab dem vierten Buchstaben nicht mehr. Ein ähnliches Ergebnis wäre mit einem noch trivialeren Ansatz erreicht worden, welcher aber weniger Berechnungen durchführen muss. Allerdings werden hier garantiert nur vier Buchstaben betrachtet. Jedem wird ein zusammenhängender Bereich der Wertemenge zugeordnet. Ist die Mächtigkeit des Alphabetes $A$, so ergibt sich eine benötigte Mächtigkeit der Wertemenge von $A^4$. Die Berechnung des Hashwertes geschieht folgendermaßen:

$$H(key) = \sum_{i=1}^{\leq 4} (p_i - 1) A^{4-i}$$

Da aber $A$ im Voraus feststeht, können für die $A^{4-i}$ feste Werte eingesetzt werden.

Im Gegensatz zum ersten, ist der zweite Ansatz schon ordnungserhaltend. Es ist jedoch schon ohne große Überlegung klar, dass die Eingabewerte ganz und gar nicht gut über den Wertebereich verteilt werden. Eher im Gegenteil, denn selbst für Wörter die mit $z$, $x$, oder $q$ beginnen, werden die gleichen großen Bereiche (für den ersten Buchstaben immerhin $A^3$ groß) reserviert. Daher ist eine Untersuchung über die durchschnittliche Verteilung von Buchstaben in der verwendeten Sprache notwendig. Da es sich hierbei um einen Prototypen handelt, wird die Untersuchung nur für eine einzige Sprache (Englisch) durchgeführt. Sinn ist es, selten auftretende Buchstaben, welche im Alphabet nebeneinander liegen, geschickt zusammen zu fassen. Das Ergebnis wird sein, dass Eingabewerte deutlich besser auf den Wertebereich verteilt und insgesamt weniger Kollisionen auftreten werden.

Die Untersuchung ist wie folgt durchgeführt worden. Als Textkorpus für die statistische Buchstabenanalyse wurde die komplette englische Wikipedia ver-

wandt [Wik04]. Diese liegt in Form einer SQL-Datei vor und beinhaltet daher auch sehr viele SQL-Befehle. Dies wird aber toleriert, um den Bezug zu den später verwendeten XML-Dokumenten herzustellen. Diese werden der INEX-Kollektion entnommen und sind daher rein (informations)technischer Natur [IEE04]. Alle zur Erstellung der Statistiken verwendeten Skripte befinden sich im Anhang. Für die einfachere Verarbeitung wurde die große SQL Datei (rund $1,2$GB) in kleinere Dateien aufgesplittet. Mittels der corpus.pl (Anhang A.1) wurden in diesen Dateien die Buchstaben gezählt und anschließend mit der statistic.pl eine Gesamtstatistik (Anhang A.2) erstellt.

```
sh$ split -b 2m -a 4 *.sql && rm *.sql
sh$ for i in `ls -1 | grep -ve "pl$"`; do cat $i \
> | ./corpus.pl >> corpus.log; done
sh$ cat corpus.log | ./statistic.pl
```

Das Ergebnis dieser Analyse ist in Tabelle 4.1 zu sehen. Um nun eine möglichst gute Streuung zu erreichen, werden nun immer so viele aufeinander folgende Buchstaben zusammen gefasst, dass jede Zusammenfassung mit einer Wahrscheinlichkeit von ungefähr 9 % auftritt. Der aufmerksame Leser wird feststellen, dass in der statistic.pl Buchstaben mit einer Häufigkeit von 12 % zusammengefasst werden. Rein rechnerisch ergibt eine Aufteilung in 11 (gewollte) Bereiche[5] eine Zusammenfassung zu $9,\overline{09}$ %-Gruppen. Experimente haben aber ergeben, dass 12 % und nicht 9 % Gruppen eben genau die gewünschten 11 Bereiche ergeben. Der Grund ist, dass nur Buchstaben zusammengefasst werden, deren Häufigkeit echt kleiner als 9 % sind. Oft überschreitet eine Gruppierung spätestens mit dem vierten Buchstaben den Schwellwert. Die Folge ist, dass es viele Dreier- und Zweier-Gruppen mit einer Häufigkeit von deutlich weniger als 9 % gibt. Mit der in Tabelle 4.2 aufgezeigten Zusammenfassung auf 11 Bereiche werden die Eingabewörter möglichst gleichverteilt auf den Bitindex (ordnungserhaltend) verstreut. Da alleine der Buchstabe $e$ mehr als 10 % aller Buchstaben ausmacht, ist eine Zusammenfassung von kleineren als 9 %-Blöcken wenig sinnvoll. Das Ergebnis wäre eine Erhöhung der Kollisionswahrscheinlichkeit für Wörter mit einem hohen Anteil von $e$. Würde man die Ordnung ignorieren, so würde sich eine viel bessere Verteilung erreichen lassen, doch dann wären Bereichsanfragen nicht mehr möglich.

Um den Vorteil der statistischen Untersuchung nutzen zu können, wird das Alphabet jetzt verkleinert. Jeder Buchstabe ist nun der Inhalt der linken Spalte jeder Zeile der Tabelle 4.2. Damit ist folgendes gemeint. Der erste Buchstabe des neuen Alphabetes ist $a, b$, also sowohl $a$, als auch $b$ sind der erste Buchstabe. Genauso sind $c$ und $d$ der zweite Buchstabe. Führt man dies bis zum Ende fort, so erhält man ein neues Alphabet, welches aus 11 Buchstaben besteht. Die Mächtigkeit $A$ ist also ab jetzt nicht mehr 36, sondern 11.

[5] Dies ist eine reine Design-Entscheidung, um den Speicherplatzbedarf gering zu halten.

| Buchstabe | absolute Anzahl | relative Häufigkeit in % |
|---|---|---|
| a | 100461069 | 7,572 |
| b | 19505168 | 1,470 |
| c | 41487288 | 3,127 |
| d | 43909028 | 3,310 |
| e | 135717964 | 10,229 |
| f | 24511749 | 1,848 |
| g | 26096012 | 1,967 |
| h | 48650451 | 3,667 |
| i | 90209825 | 6,800 |
| j | 3746825 | 0,282 |
| k | 10814221 | 0,815 |
| l | 52001560 | 3,920 |
| m | 33178233 | 2,500 |
| n | 99216352 | 7,479 |
| o | 85264235 | 6,427 |
| p | 28033027 | 2,113 |
| q | 1379844 | 0,104 |
| r | 78391266 | 5,909 |
| s | 76543760 | 5,770 |
| t | 99684474 | 7,514 |
| u | 34757615 | 2,620 |
| v | 12331205 | 0,929 |
| w | 19045194 | 1,436 |
| x | 3132270 | 0,236 |
| y | 19839120 | 1,495 |
| z | 1908534 | 0,144 |
| 0 | 27291163 | 2,057 |
| 1 | 20375786 | 1,535 |
| 2 | 17010245 | 1,282 |
| 3 | 9843274 | 0,742 |
| 4 | 11873434 | 0,895 |
| 5 | 9762570 | 0,736 |
| 6 | 7930467 | 0,597 |
| 7 | 9359259 | 0,705 |
| 8 | 9782356 | 0,737 |
| 9 | 13635340 | 1,027 |
| Gesamt | 1326680183 | 100 |

**Tabelle 4.1:** Ergebnis der Buchstabenanalyse

| Zusammenfassung | Auftrittswahrscheinlichkeit |
|---|---|
| a, b | 9,042 % |
| c, d | 6,437 % |
| e | 10,229 % |
| f, g , h | 6,428 % |
| i, j, k, l | 11,817 % |
| m, n | 9,979 % |
| o, p, q | 8,644 % |
| r, s | 11,679 % |
| t, u, v | 11,063 % |
| w, x, y, z, 0, 1 | 6,903 % |
| 2, 3, 4, 5, 6, 7, 8, 9 | 6,724 % |

**Tabelle 4.2:** Zusammfassung der Buchstaben

Die nächste Überlegung ist folgende. Da nur vier Buchstaben beachtet werden, sind viele Kollisionen bei gleich beginnenden Wörtern zu erwarten. Als Beispiel kann man "Datenverarbeitung", "Datenerfassung" oder "Datentransferierung" nehmen. Alle diese Wörter werden auf die gleiche Stelle im Bitindex gehasht. Werden aber nicht die ersten vier Buchstaben verwendet, stimmt die Ordnung nicht mehr überein. Um aber nicht unnötig viele Kollisionen zu provozieren, wird anstatt der Ordnungserhaltung die Ordnungsähnlichkeit eingeführt.

$$\forall a, b \in Definitionsbereich : a < b \Rightarrow (H(a) \leq H(b)) \vee (H(a) \not\gg H(b))$$

Das heißt, entweder sind zwei Eingaben nach dem Hashen in der gleichen Ordnung, oder, wenn dies nicht der Fall ist, sie liegen immernoch relativ dicht zusammen. Nun kann es passieren, dass zwei sehr ähnliche Wörter weiter auseinander gehasht werden als zwei weniger ähnliche. Um diesen Effekt gering zu halten werden zwei statt einer Hashfunktion verwendet. Beide sind ordnungsähnlich, und die Ausgaben für zwei gleiche Worte liegen nie mehr als $A^3$ Bits auseinander. Die erste Funktion betrachtet die ersten vier ungeraden Buchstaben, also 1, 3, 5 und 7, wogegen die zweite Funktion den ersten und die ersten drei geraden betrachtet, also 1, 2, 4, 6. Die Ausgaben beider Hashfunktionen werden nicht getrennt voneinander betrachtet, denn die erste ist anfälliger für weit auseinander gehashte, ähnliche Wörter und die zweite für Kollisionen von ähnlichen Wörtern. Die Informationen beider Funktionen gleichzeitig helfen diese Effekte gering zu halten. Das Einführen der zweiten Funktion wird später Auswirkungen haben, da sich der Speicherbedarf verdoppelt.

Als Zusammenfassung nun noch folgendes. Das Alphabet hat eine Mächtigkeit von 11, und somit wird die Wertemenge (und auch jeder Bitindex) eine Größe von 14.641 haben.

### 4.1.2 Histogrammindex - lokaler Index

Zur Erinnerung soll hier noch einmal erwähnt werden, dass für die Erstellung eines Histogramms das Wissen über den Definitionsbereich des zu indexierenden Attributes vorhanden sein muss. Das betrachtete Attribut sind die Wörter der XML-Dokumente. Als Definitionsbereich sollen hier aber nicht die eigentlichen Wörter dienen, sondern deren Bitrepräsentation im Bitindex. Im vorliegenden Fall ist der Definitionsbereich der Bereich der natürlichen Zahlen von 1 bis $A^4-1$ erweitert um die 0. Wobei $A$ die Mächtigkeit des zugrunde liegenden Alphabetes ist. Der Definitionsbereich wird in $A^3$ Bereiche aufgeteilt. Da von der Aussage, dass ein Dokument verhältnismäßig viele Wörter mit dem Anfangsbuchstaben $e$ beinhaltet, nicht auf dessen Inhalt geschlossen werden kann, wird auf die Berechnung der relativen Häufigkeit verzichtet, da die absolute Häufigkeit genau die gleiche Information liefert. Der lokale Index wird duch ein Histogamm mit ebensovielen Buckets repräsentiert. In jedem Bucket wird die Anzahl der im zugehörigen Abschnitt des Bitindexes auf eins gesetzten Bits angegeben. Wie aus Sektion 4.1.1 (Seite 37) hervorgeht, ist die Mächtigkeit des Alphabetes 11. Somit ergibt sich eine Bucketanzahl von 1331. In jedem zugehörigen Abschnitt können maximal 11 Bits gesetzt sein. Der Java Datentyp *byte* ist genau ein Byte groß und kann 127 verschiedene positive Werte annehmen. Da dies völlig ausreicht, verbraucht das Histogramm später den Platz von 1331 *byte*s, was 1,3kb (1331 Bytes) entspricht. Durch die Verdopplung des Bitindexes werden hier zwei Histogramme benötigt, womit der Speicherverbrauch auf 2,6kb (2662 Bytes) ansteigt.

## 4.2 Routing

Da dezentralisierte P2P-Netze starken Schwankungen in Hinblick auf Teilnehmerzahl und Netzstruktur unterliegen, können statische Routing-Verfahren ausgeschlossen werden. Es ist ein adaptives Verfahren nötig, welches sich sehr schnell an Veränderungen anpassen oder robust mit solchen umgehen kann.

Das Distance-Vector-Routing (DVR) hat den Vorteil, dass Informationen sehr schnell zum Adressaten geleitet werden. Das Problem in P2P-Netzen ist aber, dass für Anfragen nicht klar ist, welcher Host sie am besten erfüllen kann. Ein weiteres Problem liegt in dem Count-To-Infinity-Problem (Abbildung 3.2). Häufige Ein- und Austritte von Hosts machen viele Einträge in den Routing-Tabellen anderer Hosts nutzlos.

Auch das inhaltsbasierte Routing in Baumnetzen scheint nicht geeignet zu sein, da es sich in dieser Arbeit um ein dezentralisiertes, unstrukturiertes P2P-Netz handeln soll. Die Schwierigkeit besteht in der Anordnung aller Hosts zu einem kreisfreien Baum ohne globalem Wissen. Da nicht jeder Host jeden kennt, fällt alleine schon die Entscheidung, welcher Host die Rolle der Wurzel des Baumes übernimmt, schwer. Möchte ein Host das Netz betreten kann er nicht wissen,

ob alle gefundenen weiteren Hosts dem selben Baum angehören. Dieses Routing kann mit Veränderungen als Verfahren in dezentralisierten, strukturierten Netzen eingesetzt werden. Aber da hier jedes Strukturwissen fehlt, funktioniert dies nicht.

Das Histogramm-Routing scheint geeigneter zu sein. Es erweitert das lokale Wissen um einen definierten Horizont, was die Entscheidung des weiteren Routings unterstützt. Da aber der Prototyp nur als Aufsatz dienen soll und somit keinen Einfluss auf das zugrunde liegende P2P-Netz hat, muss auf den Vorteil des dynamischen Gruppierens verzichtet werden. Es können nicht, abhängig von den Inhalten der Repositorien, Verbindungen zu anderen Hosts hergestellt und Verbindungen zu Nachbarn gelöst werden. Dadurch besteht allerdings die Gefahr, dass weit entfernt liegende Informationen nicht gefunden werden können. Eigenes lokales Wissen geht in die Indizes der Nachbarn mit ein, doch hinter dem Horizont nicht mehr. Daher kann es passieren, dass ein Host zwar inmitten eines Netzes mit mehreren Millionen Teilnehmern ist, seine Anfragen aber nicht über seinen Horizont hinaus gelangen. Die einzige Hoffnung besteht in dem folgenden Szenario. Angenommen der Horizont wird fünf Hops weit definiert. Es wird davon ausgegangen, dass spätestens alle fünf Hops ein Host erreicht werden kann, der relevante Informationen liefert. Somit findet sich mindestens ein Routing-Index, der eine Route zur gesuchten Information offeriert. Trotz der zu erwartenden Probleme bei dem Histogramm-Routing zeichnet sich dieses Verfahren als das effizienteste ab.

### 4.2.1 Histogrammindex - Routing-Index

Der Routing-Index stellt die dritte Stufe des kompletten Indexes dar, er baut auf den Ergebnissen der lokalen Indexierung auf. Um diesen für einen Nachbarknoten zu erstellen, werden alle lokalen Indizes aller Hosts im Horizont über den Nachbarn eingesammelt. Wie groß der optimale Horizont ist, werden spätere Tests ergeben. Berechnet wird der Routing-Index wie in Definition 3.4.1 (Seite 32) beschrieben. Das Ergebnis ist, dass in jedem Bucket des Indexes eine rationale Zahl steht. In Java werden diese Zahlen durch den Datentyp *float* (und *double*) repräsentiert. Da *float* 4 Bytes groß ist, die lokalen Histogrammindizes eine Bucketanzahl von 1331 haben und durch die zweite Hashfunktion zwei Routing-Indizes nötig werden, ergibt sich ein Gesamtspeicherplatzbedarf von $10,4$kb.

Genau genommen ist die Größe des Routing-Indexes belanglos, denn er wird einmal generiert (ab und an natürlich auch erneuert) und lokal abgespeichert. Für jede Generierung müssen aber alle Hosts innerhalb des Radius ihre lokalen Histogrammindizes schicken, und jeder von ihnen ist 2662 Bytes groß. Je größer das Netz wird, desto länger braucht es, bis die lokalen Indizes übertragen werden. Damit ist abzusehen, dass das Netz ab einer bestimmten Größe unbrauchbar wird. Wenn der schlimmste Fall der ist, dass alle Hosts mittels 56

kbit/s Leitungen an das Netz angeschlossen sind, und ein Host über jeden seiner Nachbarn 1000 weitere Host im Horizont findet, so muss er für jeden Routing-Index mindestens 2, 5MB empfangen. Es ist vorstellbar, dass der Host so kaum noch Zeit hat, sich um andere Dinge als das Beantworten oder Weiterleiten von Lokal-Index-Anfragen zu kümmern. Verringert man die Anzahl der Bereiche auf $A^2$, in Folge also 121 Buckets, so müsste der Host nur noch 236kb empfangen. In diesem Fall ist aber die Aussagekraft der einzelnen Buckets geringer. Es bestehen somit zwei unvereinbare Gegensätze. Sowohl die Erhöhung der Aussagekraft des Indexes als auch die Verringerung der Größe deselben. Tests müssen hier eine geeignete Konfiguration aufzeigen.

Ein Fakt, der dem Leser eigentlich nicht entgangen sein kann, ist der, dass die Suche nach einzelnen Wörtern nicht richtig unterstützt wird. Und tatsächlich liefert die Abschätzung, dass über einen Nachbarn XML-Dokumente mit vielen Wörtern zwischen zum Beispiel 'highness' und 'hopeful' zu finden sind, keine Hilfe, wenn der Nutzer nach 'homeaddress' sucht. Wenn man aber davon ausgeht, dass der Inhalt der XML-Dokumente jeweils ein spezielles Sachgebiet beschreibt, so kann es sein, dass in einem Dokument, oder gar auf dem gesamten Host, sehr viele ähnliche Wörter vorkommen. Die Annahme ist, dass Autoren, die die gleiche Sache beschreiben (gleiches Themengebiet), viele gleiche Ausdrücke/Wörter verwenden werden. Liegen diese Dokumente auf benachbarten Hosts, wird dies eventuell in den Routing-Indizes sichtbar. Genaueres werden die Tests ergeben.

## 4.3 Anfrageverarbeitung

Die getroffene Indexierungsart bietet offensichtlich keine Unterstützung für Anfragen mit Pfadangaben. Es könnten speziell für diese Art von Anfragen ein zweiter Bitindex sowie ein zweiter Histogramm-Index eingeführt werden, welches allerdings ein global-globales Schema voraussetzt. Das Problem ist sonst, dass semantisch völlig gleiche Pfade verschiedene Namen haben können. Genau genommen basieren diese Probleme auf Synonymen, Abkürzungen und speziellen Identifikatoren. Um in einem XML-Dokument die Angestellten eines Lokales zu adressieren, gibt es unendlich viele Möglichkeiten, wie:

```
.../Ober/...
.../Kellner/...
.../Angestellter/...
.../Kell/...
.../092392/...
```

Die Grundannahme ist aber, dass kein global-globales Schema (vgl. Szenariobeschreibungen auf Seite 7) vorliegt. Somit ist der Aufbau eines allgemein verständlichen Pfadindexes nicht lösbar. Systeme wie Piazza versuchen dies mittels semantischer Übersetzung der Anfragen auf die lokalen Schemata zu lösen

[TIM+03]. Um aber nicht ganz auf Pfadanfragen verzichten zu müssen, soll eine Art Pfadausschluss eingeführt werden. Bei Anfragen der Art

```
//doc/chapter[5]/section[2]/para[@type="warning"]
```

werden nicht ausschließlich die Dokumente beachtet, die diesen Pfad beinhalten, sondern die Dokumente, welche alle Teile des Pfades in der Anfrage im Volltext beinhalten. Dafür werden alle die Dokumente ignoriert, die wenigstens einen Teil des Pfades nicht im Volltext haben. Es muss aber beachtet werden, dass die Wahrscheinlichkeit besteht, dass Dokumente durchsucht werden, die eigentlich nicht hätten durchsucht werden müssen. Der Grund ist, dass sie die geforderten Pfade nicht beinhalten, dafür aber alle Teile der Pfade im Text verteilt auftauchen.

Um die Forschung der Universität Rostock zu unterstützen, sollte als XML-Datenspeicher und Information Retrieval System eigentlich das hiesige XIRCUS-Projekt dienen, aber ob des aktuellen Entwicklungsstandes läuft dieses zur Zeit wenig stabil [MBHW02]. Daher wird als zugrunde liegendes System Berkeley DB XML verwendet [Inc05]. Alle Anfragen werden in den vom System verstandenen Sprachen XQuery 1.0 und XPath 2.0 formuliert und müssen in eine eigene Universalsprache übersetzt werden [BCF+04, BBC+05]. Da aber sowohl XQuery als auch XPath 2.0 bisher nur als Drafts erschienen sind, wird der Prototyp nur Anfragen, welche in XPath 1.0 gestellt sind, akzeptieren. Die eigene Sprache wird eine sehr einfache sein. Sie muss ausschließlich Volltextanfragen unterstützen. Als Operatoren werden nur *AND* und *OR* zugelassen. Auch wenn die beiden verwendeten Indexierungsformen Bereichsanfragen zuließen, so machte dies die Volltextsuche unmöglich. Anhand eines Beispiels kann dies deutlich gemacht werden. Sucht man mit der Suchmaschine Google [SBP+05] nach Werken von Johann Wolfgang von Goethe, die er zwischen 1776 und 1804 geschrieben hat, so wird man dies selbst im Expertenmodus nicht bewerkstelligt bekommen. Der Grund ist, dass es hierfür keinen Zugriffspfad gibt. Die Struktur der zu indexierenden Webdokumente ist nicht bekannt (und schon gar nicht dessen Inhalte), somit kann auch kein Zugriffspfad auf einen speziellen Teil der Dokumente angelegt werden[6]. Genauso verhält es sich auch bei dem Prototyp. Daher ist es nicht notwendig, Bereichsanfragen zu unterstützen.

Die Grammatik von XPath ist allgemein bekannt [CD99]. Daher kann mit Parsergeneratoren relativ schnell ein Parser generiert werden, welcher aus den XPath Anfragen alle die Teile extrahiert, welche für die eigene Anfragesprache relevant sind. Die boolschen Verknüpfungen der einzelnen Anfragekonstrukte dürfen bei dieser Prozedur nicht verloren gehen. Die so übersetzte Query kann nun gegen die eigene Indexstruktur gefragt werden. Diese Abfragen die-

[6]Es gibt ganz sicher Zugriffspfade. Diese kommen aber aus dem Bereich des Information Retrieval. Beschrieben wurden hier allerdings Zugriffspfade, wie sie aus Relationalen Datenbanken bekannt sind.

nen nur der Routing-Entscheidung. Auf den einzelnen Hosts wird dann wieder die XPath-Anfrage gegen das zugrunde liegende System gestellt.

# Kapitel 5

# Umsetzung des Prototypen

In diesem Kapitel soll der implementierte Prototyp vorgestellt werden. Es wird auf die zugrunde liegenden Techniken eingegangen und die einzelnen Bestandteile des Gesamtsystems beschrieben. Der Prototyp ist in Java 1.4 implementiert, lässt sich derzeit aber nur in Linux-Systemen verwenden. Der Grund dafür wird in Sektion 5.2.7 erklärt. Der Prototyp kann auf verschiedenen, weit verteilten Computern eingesetzt werden, wobei sich alle Instanzen des Prototypen zu einem Overlay-P2P-Netzwerk zusammen schließen. Grundlage ist hierfür Tapestry [ZKJ01, ZHS+04].

## 5.1 Tapestry

Tapestry ist ein Lokalisierungs- und Routingsystem, welches seine Routing-Entscheidungen über die Netzstruktur mittels Hashtabellen trifft. Will man es klassifizieren, so muss man es in die Klasse der dezentralisierten, strukturierten P2P-Netze einordnen. Die Arbeit bezieht sich aber auf dezentralisierte, *un*strukturierte P2P-Netze. Somit scheint die Verwendung von Tapestry hier unangebracht. Die Lösung basiert auf einem strengen Verzicht aller von Tapestry zur Verfügung gestellten Routing-Hilfen. Mittels der Hashtabellen ist es möglich, ohne Kenntnis des Netzes Nachrichten konkret an einzelne Hosts zu senden, dafür sorgt intern ein Routing-Protokoll nach Plaxton [PRR97]. Der Prototyp verwendet Tapestry als Plattform mit P2P-Funktionalitäten, hat aber alle Routing-, Indexierungs-, und Lookupverfahren selbst implementiert. Wobei mit Lookup das Auffinden von XML-Dokumenten im P2P-Netz gemeint ist.

Tapestry ist eine in Java implementierte Routing-Architektur und selbstorganisierende Infrastruktur [ZKJ01]. Normalerweise arbeiten verschiedene Java-Threads auf synchronisierten Objekten, falls die Bearbeitung der selben Objekte von Nöten ist. Die Threads (hier Stages genannt) bei Tapestry kommunizieren auf eine andere Art und Weise miteinander. Grundlage der Stage-Kommunikation ist die asynchrone I/O Bibliothek SEDA/Sandstorm [WCB01]. Auf dieser aufbauend, implementiert Tapestry ein ereignisorientiertes Program-

miermodell. Jede Stage wird als eine eigenständige Klasse implementiert, welche über einen Eventhandler verfügt. Um mit anderen Stages zu kommunizieren, wobei es egal ist, ob diese sich auf dem selben Computer oder auf einem entfernten befinden, löst die Stage ein Ereignis aus. Diese Ereignisse sind Träger für Java-Objekte, welche von einer anderen Stage weiter verarbeitet werden sollen. Es gibt zwei Arten von Ereignissen, Events und Messages. Um den Unterschied zu verstehen, ist es notwendig die Struktur einer Tapestry-Anwendung zu kennen. Die Anwendung besteht aus einer Vielzahl von Stages. Auf jedem Host im Netzwerk befinden sich die gleichen Stages. Ein Event dient als Kommunikation zwischen Stages der eigenen Anwendung und Messages zwischen Stages verschiedener Anwendungen. Es muss dazu gesagt werden, dass es möglich ist, mehrere Instanzen der gleichen Anwendung auf einem Computer zu starten. Jede Instanz fungiert als selbstständiger Host im Netzwerk. Somit sind Messages auch nötig, wenn sich die Ziel-Stage zwar auf dem selben Computer, aber in einer fremden Instanz befindet. Aus diesem Grund wird von nun an immer 'Instanz' geschrieben, wenn eigentlich 'Host' gemeint ist.

Alle auf einem Computer ausgeführten Stages laufenden parallel in einer einzigen Java-Virtuellen-Maschine (JVM). Zu jeder Instanz gehören eine Reihe von Tapestry-Verwaltungs-Stages, welche wichtige Aufgaben im Hintergrund übernehmen. Zu ihnen gehören die Network-Stage, die StaticTClient-Stage, die DynamicTClient-Stage und die Router-Stage [ZKJ+03]. Zusätzlich läuft in der JVM ein Dispatcher-Monitor. Er verwaltet alle eingehenden Events und Nachrichten und leitet diese an alle Stages weiter, die sich für genau diese Ereignisse registriert haben. Um eine Tapestry-Instanz zu starten, ist eine XML-ähnliche Konfigurationsdatei notwendig. In ihr werden für jede Stage, unter anderem der Name der selbigen, die dazugehörige Javaklasse und die zu übergebenden Parameter festgelegt. Die Konfiguration für eine Router-Stage kann beispielsweise so aussehen:

```
<Router>
  class ostore.tapestry.impl.Router
  queueThreshold 1000
  <initargs>
    pkey PublicKey1
    skey PrivateKey1
    dynamic_route static
  </initargs>
</Router>
```

Wird eine Instanz gestartet, beginnen die Verwaltungs-Stages mit dem Aufbau der Routingtabelle und der Integration in das Netzwerk. Ist diese Arbeit abgeschlossen, werden die eigenen Stages gestartet. Sind alle Stages gestartet und befinden sich in dem Zustand der Ereignisschleife, sendet die TClient-Stage ein TapestryReadyMsg-Event, worauf alle Stages mit ihrer eigentlichen Arbeit

beginnen können.

## 5.2 Prototyp-Stages

In dieser Sektion werden alle Stages beschrieben, die notwendig sind, um die geforderten Funktionalitäten zu gewährleisten. Die Stages implementieren die in den Sektionen 4.1.1, 4.1.2, 4.2.1 und 4.3 beschriebenen Verfahren beziehungsweise stellen notwendige Dienste zur Verfügung.

### 5.2.1 Center-Stage

Die Center-Stage ist die zentrale Verwaltungsstelle der selbst implementierten Stages. Die Aufgaben sind nicht sehr viele, doch sind diese wichtige Koordinationsaufgaben. Die Stage ist für vier verschiedene Events registriert. Dies sind TapestryReadyMsg, LocalIndexReadyEvent, RoutingTable$Available und TapestryRoutingTableChanged.

In der Sektion 5.1 wurde erwähnt, dass das Empfangen des TapestryReadyMsg-Events den Beginn der eigenen Arbeit einläutet. Sobald dieses Event empfangen worden ist, ist die Instanz ein vollwertiger Knoten im Netzwerk, doch in Wirklichkeit ist sie dafür noch gar nicht bereit, denn weder existieren bis dato lokale noch externe Indizes. Daher ist die Center-Stage die einzige Stage, die dieses Event empfängt. Als Reaktion wird ein ParameterInitializeEvent-Event verschickt, was dazu führt, dass die Indexer-Stage beginnt, das lokale Repositorium zu indexieren. Auch die QueryStage empfängt dieses Event. Diese beginnt aber nicht mit ihren Aufgaben, sondern initialisiert nur einige Parameter.

Ist die Indexer-Stage mit ihrer Arbeit fertig, löst sie das Ereignis LocalIndexReadyEvent aus, welches auch nur von der Center-Stage empfangen wird. Die Instanz ist nunmehr in der Lage, ihren lokalen Index auf ihre Netznachbarn zu propagieren, damit diese ihre Routing-Indizes aktualisieren beziehungsweise neu erstellen können. Dazu wird das Event LocalIndexPropagateEvent ausgelöst, welches dann von der HorizonManager-Stage empfangen wird. Gleichzeitig wird, um die Aktualität des lokalen Indexes zu gewährleisten, ein erneutes Versenden des ParameterInitializeEvent-Events geplant. Wie weit dies in der Zukunft liegt, wird vom Nutzer in der Konfigurationsdatei angegeben.

Nachdem die Verwaltungs-Stages die Instanz in das Netzwerk integriert haben lösen sie das RoutingTable$Available-Event aus. Dazu erhält die Center-Stage eine Liste mit allen direkten Nachbarn. Diese wird anschließend via NeighboursChangedEvent-Event an die HorizonManager- und die RIGeneration-Stage weitergeleitet. Im Falle des TapestryRoutingTableChanged-Events wird dieses Event um eine weitere Information erweitert. Diese besagt, welcher Nachbar das Netz verlassen oder welcher Host das Netz als Nachbar betreten hat.

### 5.2.2 Indexer-Stage

Die Indexer-Stage hat genau zwei Aufgaben, sie soll von Zeit zu Zeit den lokalen Index erneuern und die Datenbank mit dem lokalen Repositorium abgleichen. Sie wird solange untätig bleiben, bis sie das Event StartLocalIndexingProcess-Event empfängt. Ist dies geschehen, wird rekursiv der in der Konfigurationsdatei angegebene Verzeichnisbaum nach XML-Dokumenten durchsucht. Jede der gefundenen Dateien wird geöffnet, eingelesen und von Stop-Wörtern befreit. Anschließend werden die übrig gebliebenen Wörter mittels eines Porter-Stemmers auf ihre Stammform reduziert. Es wird davon ausgegangen, dass es sich bei den XML-Dokumenten um Dokumente der englischen Sprache handelt, darum wird nur eine reine englisches Stammwortreduktion verwendet. Jedes reduzierte Wort wird dann sofort mittels zweier verschiedener, ordnungsähnlicher Hashfunktionen in zwei Bitindizes gehasht.

Sind alle XML-Dokumente indexiert, werden zwei Histogramme erstellt, eines für jeden Bitindex. Jedes Feld im Histogramm repräsentiert die Anzahl der auf 1 gesetzten Bits im dazugehörigen Abschnitt im Bitindex. In Abbildung 5.1 ist dieser Vorgang verbildlicht.

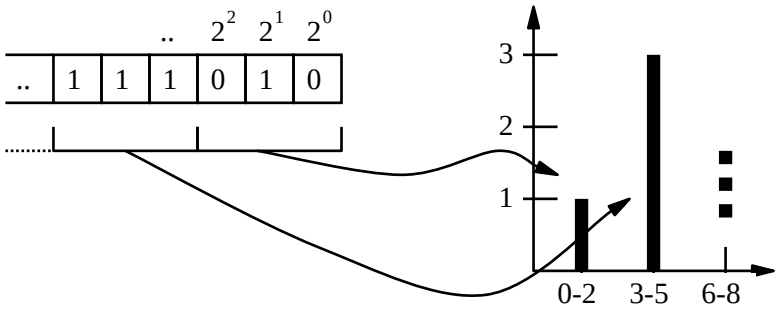



**Abbildung 5.1:** Lokale Histogrammberechnung: Wenn jedem Bucket im Histogramm ein Abschnitt von drei Bits im Bitindex zugeordnet wird, dann ergibt sich der Wert des Buckets aus der Anzahl der im Bitindex-Abschnitt gesetzten Bits.

Außerdem wird das aktuell bearbeitete Dokument, wenn nötig, in der lokalen Datenbank gespeichert.

Anschließend werden die beiden Histogramme via LocalIndexReadyEvent-Event an die Center-Stage gesandt.

### 5.2.3 HorizonManager-Stage

Die HorizonManager-Stage unterstützt alle Nachbarn innerhalb des Horizontes bei der Erstellung ihrer Routing-Indizes. Für diese ist es notwendig, alle lokalen

Indizes aller Nachbarn im Horizont zu kennen. Die Stage nimmt ihre Arbeit auf, sobald sie das LocalIndexPropagateEvent-Event von der Center-Stage empfangen hat. Dies ist das Signal, dass die Indexer-Stage ihre Arbeit beendet hat. Außerdem wartet die Stage auf das NeighboursChangedEvent-Event, denn dies beinhaltet die aktuelle Liste der Nachbarn, und mit dem Erhalt wird die veraltete Nachbarliste überschrieben.

Die Stage hat mit einem ganz bestimmten Problem zu kämpfen. In dem Moment, wo die Instanz erfolgreich gestartet ist und die Center-Stage der Indexer-Stage den Auftrag gibt, mit dem Indexieren zu beginnen, erhalten alle direkten Nachbarn das TapestryRoutingTableChanged-Event und versuchen nun, einen Routing-Index für diesen Nachbarn zu erstellen. Da aber diese Instanz noch über keinen lokalen Index verfügt, können BuildRoutingIndexMessage-Anfragen anderer Instanzen nicht verarbeitet werden. Für diesen Zweck löst die HorizonManager-Stage ein HorizonManagerPendingMessageEvent-Event aus, deren Bestandteil eben diese BuildRoutingIndexMessage-Nachrichten sind. Das Event wird aber nicht sofort ausgelöst, sondern mit einer Verzögerung von zwei Sekunden. So wird solange verfahren, bis das LocalIndexPropagateEvent-Event eingetroffen ist.

Der eigentliche Arbeitsablauf besteht aus dem Verarbeiten von BuildRoutingIndexMessage-Nachrichten in verschiedenen Stadien. Die Stadien beschreiben den Weg, auf dem sich die Nachricht befindet, sowie den Bekanntheitsgrad derer. Die Nachrichten sind ...

- **auf dem Hinweg und unbekannt:** Dies bedeutet, dass die aktuelle Instanz sich innerhalb des Horizonts der nachrichtinitiierenden Instanz befindet. Wenn eine BuildRoutingIndexMessage-Nachricht erstellt wird, bekommt sie einen global eindeutigen Identifikator. Dieser setzt sich aus der Instanz-ID und einer fortlaufenden Nummer zusammen. Die Nachricht ist deswegen unbekannt, weil der Identifikator nicht im Kurzzeitspeicher der Stage vorhanden ist. Dieser arbeitet nach dem FIFO-Prinzip, und somit werden sehr alte Nachrichten einfach wieder vergessen. Außerdem beinhaltet die Nachricht die Information, ob sie das Ende des Horizonts schon erreicht hat oder nicht. Dies ist in der vorliegenden Situation noch nicht der Fall. Darum wird der Identifikator in den Kurzzeitspeicher übernommen und die Nachricht an alle Nachbarn (außer dem Sender) weiter geleitet. Dazu wird der Streckenzähler der Nachricht um einen Wert dekrementiert. Außerdem speichert die Stage, an welche Instanzen die Nachricht gegangen ist, um später den Rückweg koordinieren zu können.

- **auf dem Hinweg und bekannt:** Dies deutet auf einen Kreis im P2P-Netz hin. Das ist nichts Schlimmes, eher im Gegenteil, denn dies garantiert die Stabilität des Netzes. Nur, wenn das Netz auch Krei-

se enthält, ist es möglich, ein Objekt auf verschiedenen Wegen von einem Punkt im Netz zu einem anderen zu leiten. Andererseits verfälscht diese Situation den Routing-Index der initiierenden Instanz, denn es werden mehr Instanzen und somit mehr Informationen in dieser Richtung des Horizonts vorgegaukelt. Um dies zu verhindern, sendet die Stage eine BuildRoutingIndexDuplicateMessage-Nachricht an den Sender zurück, damit dieser die aktuelle Instanz aus seiner Versandliste entfernt.

- **am Endpunkt und unbekannt:** Dass die aktuelle Instanz das Ende des Horizonts ist, erkennt die Stage daran, dass der Streckenzähler der Nachricht null erreicht hat. Auch hier wird der Nachrichtenidentifikator in dem Kurzzeitspeicher gespeichert. In der Nachricht wird das Erreichen des Endes markiert und der lokale Index der Instanz beigefügt. Anschließend wird sie zurück zum Sender gesandt.

- **auf dem Rückweg und bekannt:** Die Stage hatte eine Nachricht, die auf dem Weg zum Ende des Horizonts war, gespeichert und anschließend mehrere von ihnen versandt. Daher muss die Stage warten, bis alle diese Nachrichten wieder zurück gekommen sind, bevor sie die gespeicherte ihrem Vorgänger zurück schickt. Für diesen Fall besitzt die Stage einen Output-Puffer. Ist die zurückkommende Nachricht in diesem noch nicht enthalten, so wird sie dort eingefügt und der Sender aus der Versandliste für diese Nachricht gelöscht. Ist die Nachricht dort schon enthalten, wird sie um die Informationen der gerade empfangenen erweitert. Ist die Versandliste dieser Nachricht leer, sind keine weiteren Nachrichten zu erwarten, also wird die Nachricht um den eigenen lokalen Index erweitert, aus dem Output-Puffer entfernt und in Richtung des Initiators versandt. Wenn sich vorher aber herausstellt, dass der Adressat gleichzeitig der Initiator ist, wird anstatt die Nachricht zu verschicken, eine BuildRoutingIndexRIGenMessage-Nachricht erzeugt, die mit den Informationen der (BuildRoutingIndexMessage-)Nachricht angereichert und versandt wird. Diese wird dann aber nicht von der HorizonManager-Stage empfangen, sondern von der RIGeneration-Stage, die aus den empfangenen Informationen einen Routing-Index berechnet.

Alle anderen Kombinationen sind Fehlersituationen und sollten nicht auftreten können.

### 5.2.4 RIGeneration-Stage

Die RIGeneration-Stage hat als einzige Aufgabe, für jeden Nachbarn der Instanz einen Routing-Index zu erstellen. Für diesen Zweck wartet die Stage auf

NeighboursChangedEvent-Events, die von der Center-Stage versendet werden. Diese Events beinhalten immer die komplette aktuelle Liste der Nachbarn und im Falle, dass sich die Liste geändert hat, sogenannte Delta-Informationen. Diese geben an, welche Einträge in der Liste neu sind beziehungsweise welche nunmehr fehlen. Fehlen die Deltainformationen, so ist die Instanz soeben dem P2P-Netz beigetreten. Die Stage generiert für jeden Nachbarn eine BuildRoutingIndex-Message-Nachricht und setzt den Streckenzähler auf den definierten Radius des Horizonts. Sind Delta-Informationen enthalten, so werden nur diese verarbeitet. Fehlt ein Nachbar, wird sein Routing-Index aus der Liste der Routing-Indizes gelöscht und ein RoutingIndexUpdateEvent-Event initiert. Dies benötigt die Query-Stage, damit fortan dieser Nachbar ignoriert wird. Ist ein Nachbar hinzugekommen, wird nur an ihn eine BuildRoutingIndexMessage-Nachricht gesendet.

Jeder Nachbar, der eine BuildRoutingIndexMessage-Nachricht bekommen hat, wird früher oder später mit einer handleBuildRoutingIndexRIGenMessage-Nachricht antworten. Diese beinhaltet alle lokalen Indizes aller Instanzen innerhalb des Horizonts über den Nachbarn. Nun werden zwei Routing-Indizes (da jede Instanz über zwei lokale Indizes verfügt), wie in Definition 3.4.1 beschrieben, erstellt. Die erstellten Indizes werden via NewRoutingIndexAvailableEvent-Event an die Query-Stage versandt, damit diese ihre Routing-Index-Liste für diesen Nachbarn aktualisieren kann.

Es kann passieren, dass eine Instanz innerhalb des Horizonts, welche nicht direkter Nachbar ist, das Netz verlässt. Dann bekommt die Center-Stage kein Event, dass sich die (Tapestry-)Routing-Tabelle geändert hat. Aber die Routing-Indizes von mindestens einem Nachbarn sind nun nicht mehr korrekt. Um das Netz nicht mit ständigen Statusabfragen zu überfluten, wird die RIGeneration-Stage in einem in der Konfigurationsdatei festgelegten Intervall alle Routing-Indizes erneuern. Dies impliziert, dass schnell ein- und austretende Instanzen zum einen, glücklicherweise, kein Bombardement von BuildRoutingIndex-Message-Nachrichten zur Folge haben, zum anderen aber auf vielen Hosts zunächst inkorrekte Routing-Indizes hinterlassen.

### 5.2.5 Query-Stage

Die Query-Stage übernimmt alle Arten der Anfrageverarbeitung. Sie stellt Anfragen an das zugrunde liegende Text-Retrieval-System (Information Retrieval-Systeme, Datenbanken, ...) und leitet, abhängig von den Routing-Indizes, die Anfragen an andere Instanzen weiter.

Die Arbeit der Query-Stage beginnt mit dem ersten Erhalten eines New-RoutingIndexAvailableEvent-Events. Dies wurde von der RIGeneration-Stage ausgelöst und besagt, dass ab nun für mindestens einen Nachbarn ein Routing-Index vorliegt. Die Arbeit kann jederzeit von der Indexer-Stage durch ein DatabaseInUseEvent-Event unterbrochen werden. Dies kann als Datenbank-

Lock verstanden werden und dient dem exklusiven Datenbankzugriff.

Die Query-Stage wartet hauptsächlich auf zwei Arten von Ereignissen, das QueryInitiateEvent-Event, welches von der lokalen Shell-Stage ausgelöst, und die QueryMessage-Nachricht, welche von anderen Query-Stages versandt wird.

Die Reaktion auf ein QueryInitiateEvent-Event ist zunächst einmal das Anfragen der lokalen Datenbank. Reicht das erhaltene Ergebnis nicht aus, muss für die weitere Verarbeitung die XPath-Anfrage in die zum Prototyp gehörende Anfragesprache übersetzt werden. Diese ist notwendig, um eine Anfrage gegen die eigenen Indexstrukturen zu testen. Außerdem hat diese Vorgehensweise den Vorteil, dass neue Anfragesprachen leicht durch Hinzufügen eines Parsers für die konkrete Sprache unterstützt werden können. Der eigene Index verrät nun, welcher Nachbar voraussichtlich das beste Ergebnis für die Anfrage liefern wird. Es wird nun eine QueryMessage-Nachricht erzeugt und an diesen Nachbarn verschickt.

Eine erhaltene QueryMessage-Nachricht kann verschiedene Aktionen erzwingen. Dies ist abhängig von dem Status der Nachricht, die diese bei der Query-Stage genießt. Wenn eine Nachricht ...

- ... auf dem Rückweg ist, und der Ursprung der Nachricht die Query-Stage selber war, dann wird das Ergebnis der inkludierten Anfrage mittels einem ShellQueryResultEvent-Event an die Shell-Stage weitergegeben.
- ... auf dem Rückweg, aber noch nicht am Ursprung angekommen ist, dann wird die Nachricht an den Nachbarn geschickt, welcher laut in der Nachricht inkludiertem Backtrace der Sender war.
- ... zwar nicht auf dem Rückweg, aber die Lebenszeit (Hops-to-Live) abgelaufen ist, so wird, abhängig von ihrem Bekanntheitsgrad, eventuell eine Anfrage an die lokale Datenbank gestellt und sie anschließed auf jeden Fall zurück geschickt.
- ... nicht auf dem Rückweg ist und noch Lebenszeit übrig hat, so wird, abhängig von ihrem Bekanntheitsgrad eine Anfrage an die lokale Datenbank gestellt und sie anschließend zu dem Nachbarn geschickt, welcher anscheinend die besten Ergebnisse für die Anfrage liefern wird.

Grundsätzlich gilt aber, wenn nach der lokalen Anfrage die Ergebnismenge groß genug erscheint, dann wird ein weiteres Verarbeiten der Anfrage auf anderen Hosts unterdrückt und die Nachricht zu ihrem Ursprung zurück gesendet.

### 5.2.6 Flooding-Stage

Die Flooding-Stage übernimmt exakt die gleichen Aufgaben wie die Query-Stage. Der Unterschied ist, dass die getroffenen Routing-Entscheidungen vor dem (weiteren) Versand einer Anfrage anders getroffen werden. Es werden keine

Informationen über fremde Instanzen verwendet um Versandentscheidungen zu treffen. Eine Anfrage wird grundsätzlich an alle direkten Nachbarn gesendet. Es werden bei jeder empfangenen Anfrage zwei Informationen eingeholt: Ist die Anfrage schon einmal angekommen und wenn nicht, hat die Ergebnismenge ihr Maximum erreicht.

### 5.2.7 Shell-Stage

Die Shell-Stage dient als Interaktionskomponente. Sie ist der Grund, warum der Prototyp in der aktuellen Version nur auf Linux Systemen lauffähig ist. Sie verwendet einen ShellReader, welcher die Bibliothek libreadline verwendet. Dies schränkt zwar die Systemverwendbarkeit ein, erhöht jedoch den Verwendungskomfort, denn so kann die Shell fast alle Funktionalitäten bieten, die auch eine "normale" Linux-Shell bietet.

Die Hauptfunktion wird sein, dass der Benutzer Anfragen an das P2P-Netz stellt. Als Anfragen werden ausschließlich XPath-Anfragen der Version 1.0 akzeptiert, dafür aber vollständig [CD99]. Dies impliziert, dass der Prototyp ausschließlich Retrieval-Systeme unterstützen kann, welche XPath-Anfragen verstehen und verarbeiten können.

Das Ergebnis einer Anfrage besteht aus einer Liste mit gefundenen XML-Dokumenten und den Hosts, auf denen sie sich befinden. Es gibt in diesem Moment keine Möglichkeit auf die gefundenen Dokumente direkt zuzugreifen. Dies ist für den Prototyp auch nicht notwendig. Um die Güte der Anfrageergebnisse zu bestimmen, ist es notwendig zu wissen, welche XML-Dokumente im Ergebnis hätten sein müssen. Dabei ist es unerheblich, auf welchem Host jedes einzelne Dokument gefunden wurde. Wichtig ist, ob ein Dokument gefunden wurde. Die Güte der Prototyp-Ergebnisse wird in Kapitel 6 beschrieben.

Eine Anfrage wird mit dem Schlüsselwort "query" begonnen. Anschließend folgt, mit einem Leerzeichen getrennt, ein XPath-Ausdruck. Es wird sofort ein QueryInitiateEvent-Event ausgelöst, welches die zur Instanz gehörende Query-Stage veranlasst, die Anfrage zu verarbeiten, und, wenn nötig, in das Netz zu propagieren. Die Shell-Stage bleibt nun solange untätig, bis die Query-Stage als Antwort ein ShellQueryResultEvent-Event auslöst. Dieses Event beinhaltet eine Liste mit allen Fundorten aller relevanten XML-Dokumente im Netzwerk.

Mit "visual" beginnen die Kommandos zur Visualisierung des P2P-Netzwerkes. Diese werden direkt mittels einem VisualisationOrdersEvent-Event an die Visualisation-Stage versandt.

### 5.2.8 Visualisation-Stage

Um das zugrunde liegende Netzwerk zu visualisieren benötigt die Visualisation-Stage zwei Schritte. Im ersten sammelt sie Informationen über alle Hosts im Netzwerk. Diese Informationen bestehen zum einen aus einer Liste von allen

Hosts und zum anderen aus einer Liste für jeden Host, in der alle direkten Nachbarn dieser vermerkt sind. Das Einsammeln dieser Informationen geschieht durch ein "Fluten" von VisualisationMessage-Nachrichten an alle direkten Nachbarn.

Mit der Zeit antwortet jeder Host im Netzwerk mit Informationen über seine direkten Nachbarn, welche von der Visualisation-Stage gespeichert werden. Anschließend werden diese verarbeitet und das Netzwerk mit Hilfe des Programmes Colossus dargestellt [Sch04]. Ein Beispiel für eine solche Darstellung findet sich in Abbildung 6.1.

# Kapitel 6

# Ergebnisse

Um zu zeigen, ob der Prototyp eine Reduzierung der Netzlast erreichen kann, ohne dabei einen großen Verlust der Qualität der Ergebnismenge mit sich zu bringen, wurden eine Reihe von Tests durchgeführt. Dass eine Minderung der Ergebnismengenqualität zu erwarten ist, kann nicht vermieden werden. Der Grund ist, dass Anfragen, die mittels Flooding durch das Netzwerk geroutet werden, innerhalb der Hops-to-Live-Radius garantiert von allen Netzwerkteilnehmern empfangen und verarbeitet werden. Somit werden auch Hosts Ergebnisse liefern, die im Vergleich zu anderen Hosts keine guten Ergebnisse zu bieten haben. Ziel ist es, die Anfragen gezielt zu den Hosts zu routen, die eine hohe Qualität der Ergebnisse vermuten lassen.

Das P2P-Netzwerk, welches den Tests zugrunde lag, war ein Verbund aus 30 Tapestry-Instanzen. Jede von ihnen simulierte einen Host im Netzwerk. Eine Visualisation des Netzwerkes ist auf Abbildung 6.1 zu sehen.

Der gesamte Datenbestand des Netzwerkes bestand aus einem fünftel der INEX-Kollektion. Dieser Bestand wurde in zwei Testreihen verschieden auf die Hosts verteilt. Während der ersten Testreihe beherbergten die Hosts disjunkte Teile des Bestandes. Dies stellte die Situation dar, dass jedes XML-Dokument genau ein einziges Mal im ganzen Netzwerk zu finden war. Es sollte beobachtet werden, ob die Anfragen gezielt zu dem Host geroutet werden, welcher das gesuchte XML-Dokument besitzt. In dem zweiten Testlauf wurden die Daten redundant gespeichert. Es sollte nun die Veränderung im Routing-Verhalten beobachtet werden.

Um die Qualität der Anfragen bewerten zu können, wurde während des Routings von jedem Anfrageobjekt eine Spurverfolgung aktiviert. Anhand dieser lässt sich später bestimmen, welchen Weg die Anfrage durch das Netzwerk genommen hat. Außerdem wurde jede Anfrage sowohl mittels des neuen Routing-Verfahrens als auch mittels Flooding verbreitet.

Da das Ergebnis der "gefluteten" Anfragen exakt den Erwartungen entsprach, soll auf diese Versuche im folgenden nicht mehr eingegangen werden. Die Ergebnismenge entsprach genau der maximal erreichbaren Ergebnismenge (alle

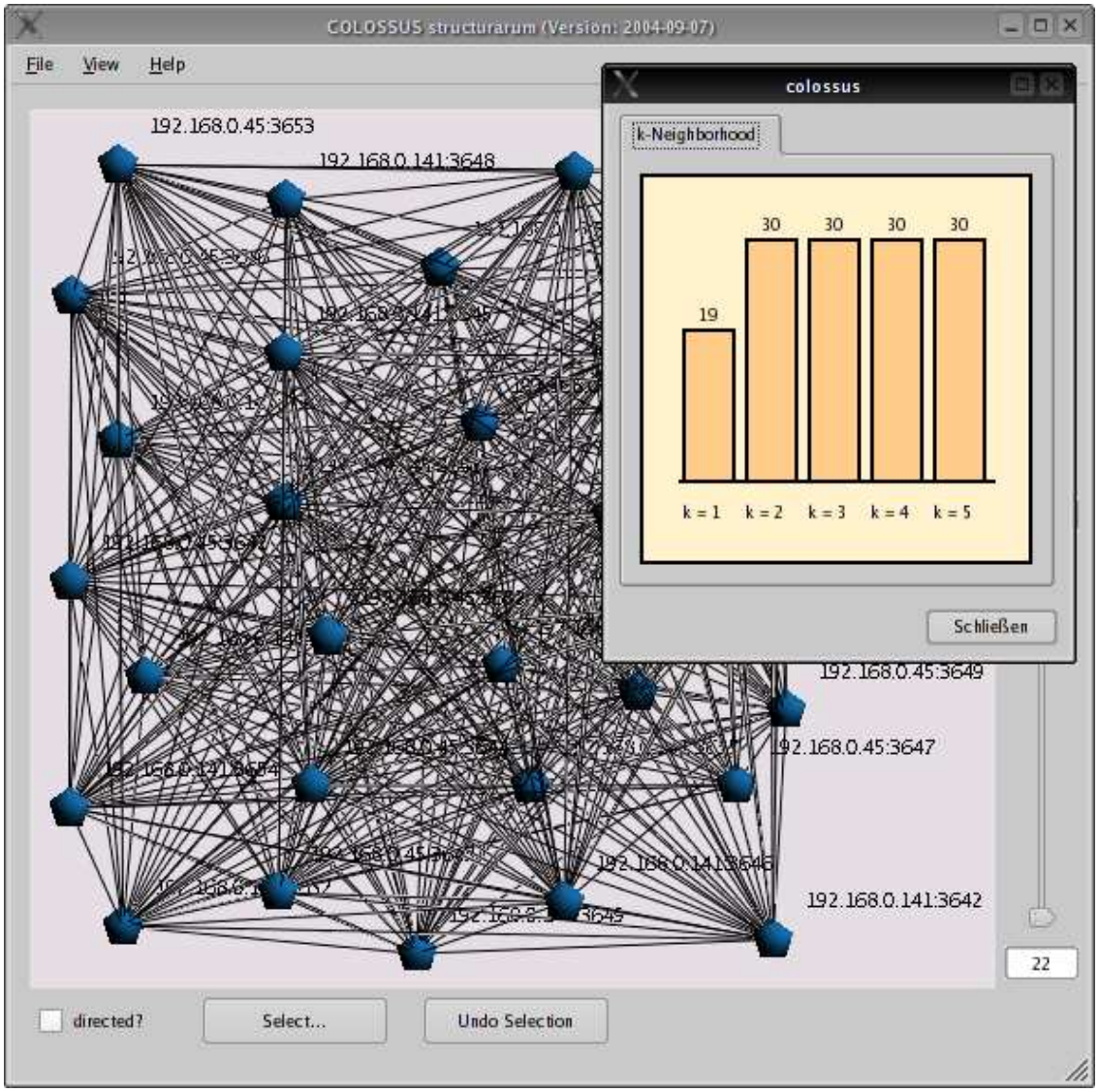



**Abbildung 6.1:** Visualisation des Testnetzes

relevanten Dokumente wurden gefunden). Wie zu erwarten, stieg bei diesen Anfragen die Netzlast derart, dass es nicht möglich war, zwei Anfragen gleichzeitig zu verschicken. Dies scheint zunächst ein Fehler in der Implementierung des Floodings zu sein, aber mit dem Hintergrund, dass die Hälfte aller Netzwerkteilnehmer jeweils auf einem physischen Rechner simuliert wurden und dass jeder Host über eine eigene Berkeley DB XML Instanz verfügt, scheint es klar, dass nicht die Netzlast das Netzwerk ausbremste, sondern die hohe Anzahl der Festplattenzugriffe.

Alle in den Tests verwendeten Anfragen waren etwa der gleichen Struktur. Es wurde nach XML-Dokumenten gesucht, die einen gewissen absoluten Pfad beinhalten und ein ganz konkretes Textelement. So wurde nach Dokumenten gesucht, welche als Hauptautor einen Menschen mit dem Nachnamen 'Ibbett' haben. Die dazugehörige Anfrage an den Prototypen sieht wie folgt aus.

```
192.168.0.141:3632>
  query /article/fm/au[@sequence="first"]/snm[text()="Ibbett"]
```

Grundsätzlich wurde im Voraus die erreichbare Ergebnismenge ermittelt und anschließend mit dem erbrachten Ergebnis verglichen.

Eine Zusammenfassung der Ergebnisse soll an einem konkreten Beispiel vorgenommen werden. Dazu wird zunächst näher auf die Einstellungen des Prototypen eingegangen. Das Netzwerk bestand aus 30 Teilnehmern, und es war möglich, in zwei Hops grundsätzlich jeden beliebigen Host zu erreichen. In der Abbildung 6.1 ist dies in dem kleinen PopUp-Fenster angedeutet. In diesem Fall besaß der Host 19 direkte Nachbarn und konnte über diese Nachbarn alle 30 Hosts erreichen. Um aber das Wissen der einzelnen Hosts nicht zum globalen Wissen verkommen zu lassen, ist der Horizont mit einer Tiefe von nur einem Hop gewählt worden. Mit der Zeit hat sich herausgestellt, dass die Lebenszeit der Anfragen ungefähr doppelt so hoch sein muss als bei "gefluteten" Anfragen. Der Grund ist, dass das prototypische Routing ein Verhalten zeigt, welches stark an *Trial-and-Error* erinnert. Bei jedem Schritt zurück wird der Streckenzähler verringert, also auch auf Pfaden, die schon einmal eingeschlagen wurden. Da die Mächtigkeit der Ergebnismenge im Voraus bekannt war, wurde die Ergebnismengengrenze auf einen Wert oberhalb der Mächtigkeit gesetzt, um das Ergebnis nicht zu verfälschen.

Die Anfrage, anhand der im folgenden die Auswertung stattfindet, sollte alle Dokumente der Kollektion finden, welche zu Tagungen gehören, die von Juli bis August andauerten.

```
192.168.0.141:3632>
  query /article/fm/hdr/hdr2/pdt/mo[text()="JULY-AUGUST"]
```

Die Ermittlung der Ergebnismenge ergab, dass insgesamt 137 Dokumente zu finden und diese auf acht Hosts verteilt waren. Keiner dieser Hosts war derjenige, von dem die Anfrage gestartet wurde. Theoretisch ist es also möglich, die

komplette Ergebnismenge nach 16 Hops gefunden zu haben (da in zwei Hops jeder Host erreichbar ist).

In den ersten Testläufen (keine redundante Speicherung) war folgendes zu beobachten. Die Anfrage wurde grundsätzlich gezielt zu einem Host geroutet, der viele Ergebnisse zu liefern hatte. Für das konkrete Beispiel wurde in zwei Hops ein Host erreicht, welcher alleine $14,6$ % der gewünschten Ergebnismenge erbrachte. Das erfolgreiche Routing endete dort, denn ob der Attraktivität des Hosts wurde die Anfrage immer wieder zu diesem Hosts zurück geroutet. Jeder Host in dem Horizont des attraktiven Hosts hatte alle anderen Träger der gesuchten Information in seinen Routing-Indizes, doch haben die direkten Nachbarn immer einen Weg gesucht, die Anfrage wieder zu dem attraktiven Host zu leiten. Dieses Verhalten erinnert an das Zeichnen einer Blume und ist in Abbildung 6.2 dargestellt. Es hat sich gezeigt, dass das Erhöhen der Le-

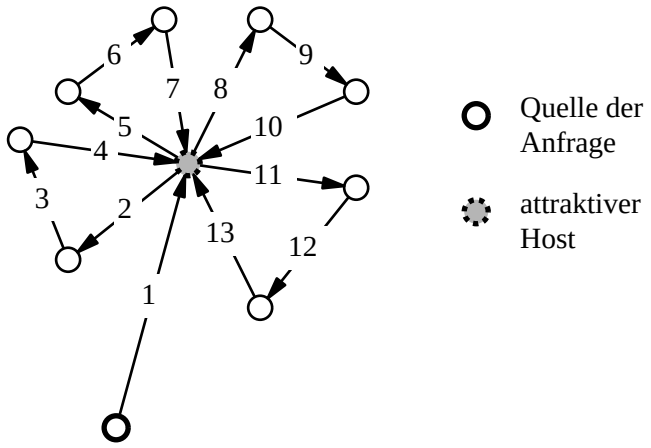



**Abbildung 6.2:** Routing-Verhalten bei Einzigartigkeit von Dokumenten

benszeit nur bis zu einem bestimmten Wert eine Verbesserung des Ergebnisses bewirkte. Im Falle des konkreten Netzes hat sich ab einer Lebenszeit von fünf Hops keine weitere Verbesserung eingestellt.

Gänzlich anders verhielt es sich in den Testläufen zweiter Art. Jedes Dokument war genau zwei Mal im Netzwerk gespeichert. Die Anfragen wurden durch das Vorhandensein gleicher Dokumente im Netz von einzelnen Hosts abgelenkt, und so erreichten sie im Durchschnitt all fünf Hops einen Host, welcher das Rückgabeergebnis positiv beeinflusste. Die Erhöhung der Lebenszeit ging mit der Erhöhung der Ergebnisqualität einher. Tabelle 6.1 zeigt diese Beobachtung. Die letzte Zeile zeigt, dass der Streckenzähler im Verhältnis zur Größe des Netzes sehr hoch initialisiert werden muss, damit Duplikate in dem erbrachten Ergebnis auftauchen. Eine weitere Erhöhung hatte keine Verbesserung der Ergebnismenge zur Folge.

| Initialwert des Streckenzählers | Gefundene Dokumente in % |
|---|---|
| 05 | 14, 6 |
| 10 | 40, 9 |
| 15 | 55, 5 |
| 20 | 55, 5 |
| 25 | 66, 4 |
| 30 | 66, 4 + Duplikate |

**Tabelle 6.1:** Ergebnisverbesserung durch Lebenszeiterhöhung

Als Zusammenfassung der Tests kann gesagt werden, dass das Einbeziehen von Repositoriumsinformationen der Hosts innerhalb einer gewissen Reichweite zunächst zu guten Routing-Entscheidungen führt, mit zunehmendem Alter der Anfragen aber immer weniger Ergebnisse liefert. Dafür ist gezeigt worden, dass durch die Vermeidung von vielen gleichzeitigen gleichen Anfragen und deren Ersetzung durch eine einzelne die Netzlast soweit sinkt, dass die Lebenszeit der Anfragen um ein Vielfaches erhöht werden kann, ohne dass die Zeit zwischen Anfrage und Ergebnis steigt.

# Kapitel 7

# Schlussbetrachtung

In diesem Kapitel wird eine Zusammenfassung der Diplomarbeit gegeben. Außerdem werden Vorschläge zur Verbesserung oder zur Weiterentwicklung gegeben.

## 7.1 Zusammenfassung

Ziel dieser Diplomarbeit war es, ein Indexierungsverfahren zu entwickeln, mit dessen Hilfe es möglich ist, gezielt Anfragen durch dezentralisierte, unstrukturierte P2P-Netzwerke zu routen.

Ein dreistufiger Index ermöglicht es, Abschätzungen über den Inhalt des Repositoriums eines Teilnehmers zu machen. Die erste Stufe sieht die Erstellung einer eindeutigen Signatur für das Repositorium vor. Die zweite ist die Generierung eines Histogramms über die Verteilung einzelner Bits in der Signatur. Das Routing wird unterstützt, indem Hosts ihr lokales Wissen um einen definierten Radius erweitern. Zu diesem Zweck sammeln sie die Indizes aus der zweiten Stufe ihrer Netznachbarn und konstruieren die dritte Stufe des Indexes. Mittels dieser sind die einzelnen Hosts in der Lage, Anfragen gezielt an ihre Netznachbarn weiterzuleiten.

Die prototypische Implementierung ist in Java erfolgt. Als Datenbanksystem liegt diesem Berkeley DB XML zugrunde [Inc05]. Jeder Host besitzt ein Repositorium von XML-Dokumenten und bietet diese anderen Hosts zu ihrer Verfügung an. Tritt ein Host dem Netzwerk bei, durchläuft er zunächst die ersten beiden Indexierungsstufen und bietet das Ergebnis allen seiner Netznachbarn an. Anschließend verlangt er die gleichen Indizes von ihnen und generiert daraus für jeden seiner direkten Netznachbarn einen Index der dritten Stufe. Verlässt ein Host das Netzwerk, werden seine direkten Netznachbarn augenblicklich alle Informationen über ihn löschen, so dass keiner versuchen wird, den nunmehr fehlenden Host zu erreichen.

Durch die erweiterten Informationen aller Hosts sind diese in der Lage, Anfragen gezielter durch das strukturlose Netzwerk zu routen und helfen dabei,

die Nutzbarkeit des Netzwerkes durch sehr geringe Netzlast zu erhöhen.

## 7.2 Ausblick

Der Prototyp sollte zeigen, dass mittels eines neuartigen Routingverfahrens die Netzlast gesenkt werden kann, ohne die Qualität der Ergebnisse stark zu beeinflussen. Das Ergebnis einer Anfrage ist eine Liste aller relevanten XML-Dokumente und deren Speicherort im Netzwerk. Diese Darstellung reichte für die Betrachtung der Ergebnisqualität völlig aus, kann aber für weitere Verwendungen derart erweitert werden, dass es eine Möglichkeit gibt, direkt auf die XML-Dokumente zuzugreifen.

Um Dokumente im Netzwerk zu finden, ist es für den Prototypen notwendig, dass sich der Inhalt der gesuchten Dokumente etwas von allen anderen Dokumenten abhebt. Sind Dokumente mit einzigartigem Inhalt im Netzwerk vorhanden, doch unterscheidet sich die Wortwahl im Inneren kaum von anderen Dokumenten, ist es für den Prototypen kaum möglich diese Dokumente zu finden. Ein Ansatz zur Lösung dieses Problems ist das Einführen von Replikationen einzelner Dokumente [Poh04]. Eventuell sollte grundsätzlich eine Replikation aller Daten erfolgen, denn Tests wie sie in Kapitel 6 beschrieben worden sind, haben gezeigt, dass die Qualität der Ergebnisse steigt, wenn Dokumente redundant im Netzwerk vorhanden sind.

In Sektion 3.4 wird eine Verbesserung des Routings durch Gruppierung von Hosts und Manipulation der Netzstruktur erreicht. Bei der Entwicklung des Prototypen ist davon ausgegangen worden, dass er ein Aufsatz auf ein bestehendes System ist und keine Möglichkeit hat, die Netzstruktur zu ändern. Wenn zwischen P2P-Netzwerk und Prototyp eine logische Netzsicht implementiert wird (ein Overlay-Netzwerk eines Overlay-Netzwerkes), kann der Prototyp innerhalb dieser Schicht die Netzstruktur ändern und somit von der Gruppierung inhaltlich ähnlicher Hosts profitieren.

Die Aussagekraft der Bitsignatur der Repositorien lässt sich erhöhen, wenn nicht das absolute Vorhandensein eines Wortes ein Bit in der Signatur setzt, sondern ein überdurchschnittliches Auftreten dessen. Dies bedeutet, dass der Prototyp auf ein ganz konkretes Anwendungsszenario abgestimmt wird und sich dann auch nicht mehr sinnvoll für XML-Dokumente anderer Themenbereiche nutzen lässt. Für diese Verbesserung muss zunächst ein Textkorpus vorliegen, der die gleiche statistische Häufigkeitsverteilung der Wörter aufweist, wie die zu erwartenden XML-Dokumente. Mittels dieses Korpus kann ein Filter angelernt werden, so dass ein Wort nur dann ein Bit in der Signatur setzt, wenn es eine Häufigkeit aufweist, die über dem im Filter definierten Schwellwert für das Wort liegt. Die Signatur gibt sodann nicht mehr den Inhalt eines Repositoriums wieder, sondern das, was das Repositorium im Gegensatz zu anderen Repositorien auszeichnet.

Die Anfragen werden von dem Prototyp in einer Trial-and-Error-Manier durch das Netzwerk geroutet. Ist die Lebenszeit der Anfrage abgelaufen, oder hat die Ergebnismenge eine zufriedenstellende Größe erreicht, so wird das Ergebnis mittels Backtracking zum Ursprung zurück geschickt. Dies bedeutet, dass auch alle Pfade, welche sich als Sackgasse herausgestellt haben, in umgekehrter Richtung durchlaufen werden. Wenn man das Routing durch ein klassisches, adaptives Routing-Verfahren erweitert, kann das Ergebnis ohne Umwege auf dem schnellsten Weg zum Ursprung gelangen. Auf dem Hinweg funktioniert diese Erweiterung nicht, da der oder die Adressaten einer Anfrage nicht bekannt sind. Wird der Streckenzähler einer Anfrage mit 30 initialisiert, und ist der Endpunkt der Anfrage ein direkter Nachbar des Ursprungs, so werden auf dem Rückweg 29 Hops eingespart.

# Anhang A

# Perl-Skripte

Hier werden verschiedene Perl-Skripte angegeben, die für die Statistikberechnungen der Buchstaben- und Wortverteilungen verwendet wurden.

## A.1 corpus.pl

```
#!/usr/bin/perl

#
# corpus.pl
# howto use: cat <file> | corpus.pl
#

undef $/;

my $gesCount=0;
my %alphas;
my $wordCount=0;
while(<>){
    foreach my $char (a..z,0..9){
        $alphas{$char}+= $_ =~ s/$char//igs;
    }
}
$gesCount=0;
foreach (sort keys %alphas){
    $gesCount += $alphas{$_};
    print "$_: $alphas{$_}\n";
}
print "Buchstaben gesamt: $gesCount\n";
```

## A.2 statistic.pl

```
#!/usr/bin/perl

#
# statistic.pl
# howto use: cat <file> | statistic.pl
#

my %gesCount;

while(<>){
    if (/^([0-9a-z ]+): (\d+)$/i){
        $gesCount{lc($1)} += $2;
    }
}

if($gesCount{"buchstaben gesamt"}){
    my %packages;
    my $relative = 0;
    my $who = "";
    my $auslastung = 0;

    foreach (sort keys %gesCount){
        my $value = $gesCount{$_}/$gesCount{"buchstaben gesamt"};
        print "$_: $gesCount{$_}    ---   relativ: $value\n";
        my $tmp = $relative + $value;
        if($tmp > 0.12){
            $packages{$who} = $relative;
            $auslastung += $relative;
            $relative = $value;
            $who = $_;
        } else {
            $relative = $tmp;
            $who .= ", $_";
        }
    }

    $packages{$who} = $relative;
    $auslastung += $relative;

    foreach (keys %packages) {
        print "$_: $packages{$_}\n";
```

```
    }
    print "gesamte auslastung: ".--$auslastung."\n";

} else {
    foreach (sort keys %gesCount){
        print "$_: $gesCount{$_}\n";
    }
}
```

# Anhang B

# ANTLR Parser Generator

Zum generieren der Java XPathParser und XPathLexer Klassen wurde der Parsergenerator ANTLR verwendet [Par05]. Die Grammatikdefinitionsdateien sind zu groß, um sie an dieser Stelle wiederzugeben. Es sei daher auf die zu dieser Arbeit gehörenden CompactDisc verwiesen. Man findet dort im Verzeichnis

`prototyp/source/xmldb.net/trunk/src/de/xmldb/net/util/`

die Grammatiken für den XPathParser (XPathParser.g) und den XPathLexer (XPathLexer.g).

# Abbildungsverzeichnis

# Liste der Algorithmen

# Tabellenverzeichnis

# Literaturverzeichnis


[BBC+05] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. *XML Path Language (XPath).* W3C, 2.0 edition, February 2005. http://www.w3.org/TR/2005/WD-xpath20-20050211/.

[BCF+04] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. *An XML Query Language (XQuery).* W3C, 1.0 edition, July 2004. http://www.w3.org/TR/2004/WD-xquery-20040723/.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath).* W3C, 1.0 edition, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.html.

[Cli] Clip2, http://rfc-gnutella.sourceforge.net/developer/stable /index.html. *The Annotated Gnutella Protocol Specification*, v0.4 edition. Document Revision 1.6.

[CS01] Brian Cooper and Moshe Shadmon. The index fabric: Technical overview. Technical report, RightOrder Inc., 2001.

[CSA+02] Dario Colazzo, Carlo Sartiani, Antonio Albano, Paolo Manghi, Giorgio Ghelli, Luca Lini, and Michele Paoli. A typed text retrieval query language for xml documents. *J. Am. Soc. Inf. Sci. Technol.*, 53(6):467–488, 2002.

[FCDH91] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order-preserving minimal perfect hash functions and information retrieval. In *ACM Transactions on Information Systems*, volume 9, pages 281–308. ACM, July 1991.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann Publishers Inc., 1997.

[Hof02] Roland Hofwiler. Das Ende der Legende. *Die Tageszeitung*, 6766:17, June 2002. TAZ, 5. June 2002, 218 Lines, Contrapress media GmbH.

[HS99a] Andreas Heuer and Gunter Saake. *Datenbanken: Implementierungstechniken*, chapter 4.3.3. MITP-Verlag GmbH, 1 edition, 1999.

[HS99b] Andreas Heuer and Gunter Saake. *Datenbanken: Implementierungstechniken*, chapter 4.4. MITP-Verlag GmbH, 1 edition, 1999.

[Huf52] David A. Huffman. A method for the construction of minimum redundancy codes. In *roceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[IEE04] Initiative for the evaluation of xml retrieval. IEEE Computer Society, 2004. http://www.ieee.org.

[Inc05] Sleepycat Software Inc. Berkeley db xml 2.0 now available! http://www.sleepycat.com/products/xml.shtml, 2005. seen 12. March 2005.

[KM03a] Meike Klettke and Holger Meyer. *XML & Datenbanken*, chapter 9.2.2. dpunkt.verlag GmbH, Ringstraße 19b, 69115 Heidelberg, 1 edition, 2003.

[KM03b] Meike Klettke and Holger Meyer. *XML & Datenbanken*, chapter 9.4.4. dpunkt.verlag GmbH, Ringstraße 19b, 69115 Heidelberg, 1 edition, 2003.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[Lit78] Witold Litwin. Virtual hashing: A dynamically changing hashing. In S. Bing Yao, editor, *Fourth International Conference on Very Large Data Bases, September 13–15, 1978, West Berlin, Germany*, pages 517–523. IEEE Computer Society, 1978.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: a database management system for semistructured data. *SIGMOD Rec.*, 26(3):54–66, 1997.

[MBHW02] Holger Meyer, Ilvio Bruder, Andreas Heuer, and Gunnar Weber. The xircus search engine. In *INEX Workshop*, pages 119–124, 2002.

[Mor68] Donald R. Morrison. Patricia–practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[Par05] Terence Parr. Antlr parser generator. http://www.antlr.org, 2005. Version 2.7.5.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[PKP04] Yannis Petrakis, Georgia Koloniari, and Evaggelia Pitoura. On using histograms as routing indexes in peer-to-peer systems. Technical report, Departmen of Computer Science, University of Ioannina, Greece, August 2004. DBISP2P 2004, Toronto, Canada.

[Poh04] Andreas Pohl. Datenverteilung in Peer-to-Peer Overlay-Netzwerken. Diplomarbeit, Universität Rostock, Lehrstuhl Datenbank- und Informationssysteme, November 2004.

[Pri01] Michael T. Prinkey. An efficient scheme for query processing on peer-to-peer networks. http://todo.edu, 2001. draft, seen 10. Februar 2005.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA*. ACM, June 1997.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[Rob86] John T. Robinson. Order preserving linear hashing using dynamic key statistics. Technical report, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598, 1986.

[Rö03] Janko Rötgers. *Mix, Burn & R. I. P.*, chapter 1, pages 22–23. Verlag Heinz Heise, 2003.

[SBP+05] Eric E. Schmidt, Sergey Brin, Larry Page, John Doerr, Michael Moritz, Ram Shriram, John Hennessy, Arthur Levinson, and Paul Otellini. Google. http://www.google.com, Februar 2005.

[Sch04] Hans-Jörg Schulz. Visuelles Data Mining komplexer Strukturen. Master's thesis, Universität Rostock, 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[Swg05] Swgreed. Fasttrack. http://de.wikipedia.org/wiki/FastTrack, January 2005. Version 12:33, 21. Jan 2005.

[Tan00] Andrew S. Tanenbaum. *Computernetzwerke*, chapter 5. Pearson Education Deutschland GmbH, Martin-Kollar-Straße 10–12,81829 München, 3 edition, 2000.

[TIM+03] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin (Luna) Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. *SIGMOD Rec.*, 32(3):47–52, 2003.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Eighteeth Symposium on Operating Systems Principles*. University of California, Berkeley, Computer Science Division, ACM, October 2001. http://www.cs.berkeley.edu/∼mdw/proj/seda/.

[Wik04] Wikipedia. http://www.wikipedia.com, December 2004. 09 Dec 2004 dump.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal On Selected Areas In Communications*, volume 22, pages 41–53, January 2004.

[ZKJ01] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing.

Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001. seen 31.Januar 2005.

[ZKJ+03] Ben Y. Zhao, John D. Kubiatowicz, Anthony D. Joseph, Kris Hildrum, Ling Huang, Sean Rhea, Jeremy Stribling, Calvin Hubble, and Ben Poon. Tapestry: Infrastructure for fault-resilient, decentralized location and routing. http://todo.edu, July 2003.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 31. März 2005

# Thesen

1 Große Nachfrage nach bestimmten Informationen führt in der klassischen Client/Server-Architektur oft zur Überlastung einzelner Netzwerkteilnehmer und zur Nicht-Erreichbarkeit ganzer Teilnetze.

2 Die Verwendung dezentralisierter, unstrukturierter Peer-to-Peer-Systeme erhöht die Verfügbarkeit des Netzwerkes durch die Verteilung von Daten auf mehrere unabhängige, gleichberechtigte Netzwerkteilnehmer. Der Ausfall einzelner Teilnehmer beeinträchtigt nicht die Funktionalität des Netzes.

3 Flooding ist das am häufigsten eingesetzte Routing-Verfahren in dezentralisierten, unstrukturierten Peer-to-Peer-Netzen. Die unnötige Vervielfältigung von Anfragen erhöht die Netzlast drastisch und vermindert somit die Nutzbarkeit des Netzwerkes.

4 Klassische Routing-Verfahren lassen sich in Peer-to-Peer-Netzwerken nicht einsetzen, da weder die Struktur des Netzwerkes noch der Adressat einer Anfrage im Voraus bekannt sind.

5 Mit ausschließlich lokalem Wissen kann kein Netzwerkteilnehmer Routing-Entscheidungen treffen.

6 Durch die Indexierung von Metainformationen über Netzwerkteilnehmer können Abschätzungen über zu erwartende Ergebnismengen getroffen werden.