

Bauhaus-Universität Weimar
Fakultät Medien
Web Technology & Information Systems

Abschlussarbeit
zur Erlangung des akademischen Grades

BACHELOR OF SCIENCE

NETSPEAK
**Ein Assistent zum Verfassen
fremdsprachiger Texte**

von

MARTIN TRENKMANN

Kaufstraße 15
99423 Weimar

Betreuer

Prof. Dr. Benno Stein
Dipl.-Inf. Martin Potthast

22. Juni 2008

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Verwendung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, habe ich als solche gekennzeichnet. Weiterhin wurde diese Arbeit noch keiner anderen Prüfungsbehörde vorgelegt oder als Teil sonstiger Prüfungsleistungen eingereicht.

Weimar, 22. Juni 2008

Martin Trenkmann

Kurzfassung

Das Verfassen von Texten in einer fremden Sprache ist auch für erfahrene Autoren eine schwierige Aufgabe. Angefangen bei Problemen der richtigen Wortwahl in einem gegebenen Kontext bis hin zu der Frage, welche Redewendung gerade am gebräuchlichsten wäre. Ein ungewöhnlicher Ausdruck oder die Verwendung falscher grammatikalischer Formen lässt dabei schnell auf die Muttersprache des Verfassers schließen.

In dieser Arbeit wird der neuartige Web-Dienst NETSPEAK vorgestellt, der Autoren in dieser Situation unterstützen soll, indem er das World Wide Web als Quelle für Ausdrucksweisen und Redewendungen nutzt. Dabei wird angenommen, dass die Häufigkeit des Auftretens einer bestimmten Wortkombination im Web repräsentativ für deren Gebräuchlichkeit im modernen Sprachgebrauch ist.

Der Dienst kann zum einen mit kurzen Textphrasen angefragt werden, um deren Richtigkeit zu überprüfen und zum anderen bietet er eine Unterstützung für sogenannte Wildcard-Queries. Eine solche Anfrage bietet die Möglichkeit an bestimmten Stellen der Query Platzhalter einzufügen, um diese durch sinnvolle Worte ersetzen zu lassen. Auf diese Weise können gängige Wortverbindungen erfragt oder Sätze vervollständigt werden. Zur Unterstützung der richtigen Wortwahl kann ein einzelnes Wort zudem für eine Synonymsuche gekennzeichnet werden. Das Ergebnis ist eine nach Häufigkeiten sortierte Liste von Vorschlägen gültiger Formulierungen.

Zur Beantwortung einer Anfrage ist es notwendig alle diese Formulierungen mit ihren Häufigkeiten zu kennen. Die Google-N-Gramm-Kollektion bietet solche Daten: Es wurden eine Billion Wort-Token von öffentlichen Web-Seiten analysiert und Folgen bis zu einer Länge von fünf Worten protokolliert. Für jede Wortfolge (N-Gramm) wurde die Häufigkeit ihres Auftretens ermittelt. NETSPEAK verwendet die rund eine Milliarde umfassende Menge der 5-Gramme zur Suche, die in einer Größe von 32 Gigabyte auf einer Festplatte vorliegt. Da der Dienst jederzeit verschiedene Anfragen erhält, müssen alle Wortfolgen schnell zugreifbar sein. Die Daten wurden daher mit einem invertierten Index indiziert. Die Konstruktion dieses Indexes wird ebenfalls in dieser Arbeit vorgestellt.

Inhaltsverzeichnis

Erklärung	iii
Kurzfassung	iv
1 Einleitung	2
1.1 Die Idee zu Netspeak	5
1.2 Information-Retrieval-Systeme	6
2 Der Web-Dienst Netspeak	9
2.1 Der Retrieval-Prozess	10
2.2 Syntax der Anfragesprache	11
2.2.1 Allgemeine Query-Grammatik	11
2.2.2 Spezielle Query-Grammatik	13
2.3 Spell-Correction	14
2.4 Query-Expansion	15
2.4.1 Synonymsuche	15
2.4.2 Mustersuche im N-Gramm-Index	16
2.4.3 Zusammenfassen von N-Grammen	18
2.5 Ranking nach Häufigkeiten	19
2.6 Visualisierung im Web-Interface	19
2.6.1 Nachladen von Textbeispielen	22
2.7 Performanz des Web-Dienstes	22
2.7.1 Einzelne Anfrage	22
2.7.2 Parallele Anfragen	24
3 Indizierung großer Datenmengen	26
3.1 Datenstruktur der invertierten Liste	27
3.1.1 Entfernung von Stoppworten	28
3.1.2 Reduktion auf den Wortstamm	29
3.2 Implementierung eines invertierten Index	30
3.2.1 Eingabeformat: Pseudo-invertierte Liste	31
3.2.2 Indexkomponenten und ihre Funktionen	34
3.2.3 Prozess der Indexkonstruktion	40
3.2.4 Suche im Index	42
3.2.5 Minimale perfekte Hashfunktion	43

3.2.6	Abschätzung zum Speicherverbrauch	47
3.2.7	Anbindung an Java	48
4	Zusammenfassung und Ausblick	50
	Literaturverzeichnis	51
A	Netspeak Service	53
A.1	Eclipse Projekt (Java)	53
A.2	Dokumentation (Javadoc)	53
B	Inverted Index	54
B.1	Implementation	54
B.1.1	Codeblocks Projekt (C++)	54
B.1.2	Eclipse Projekt (Java)	54
B.2	Dokumentation	54
B.2.1	C++ (Doxygen)	54
B.2.2	Java (Javadoc)	54

1 Einleitung

Heutzutage verfassen Schüler, Studenten und Wissenschaftler oft englische Texte und auch im täglichen Umgang mit dem World Wide Web ist Englisch für viele die dominante Sprache. Die meisten Menschen allerdings stehen in solchen Situationen vor einer großen Hürde, da Englisch nicht ihre Muttersprache ist.

Die Probleme entstehen bereits bei an sich einfachen Dingen, wie der Wahl der richtigen Präposition zwischen einem Verb und einem Substantiv oder dem Finden der richtigen Formulierung, um einen konkreten Sachverhalt zu beschreiben. Eine Orientierung an vorhandenen phonetischen Ähnlichkeiten zwischen einem Wort der Muttersprache und einer englischen Übersetzung muss dabei nicht immer richtig sein, wie folgende Beispiele veranschaulichen:

(a)

Ich bin ein Student, der ...

- ... **an** Informatik interessiert ist.
- ... sich **für** Informatik interessiert.

I'm a student, who is interested ...

- ... **at** computer science.
- ... **on** computer science.
- ... **for** computer science.

(b)

*Ich parke mein Auto **vor** dem Gebäude.*

*I park my car **before** the building.*

Während in Beispiel (a) alle Übersetzungen tatsächlich grammatikalisch falsch sind (richtig wäre: ... *in computer science*), spielt in Beispiel (b) der Kontext die entscheidende Rolle. Das Adverb *before* ist in diesem Zusammenhang falsch, da es zur Beschreibung zeitlicher Vorgänge dient. Eine entsprechende physikalisch gegebene Konstellation wird hingegen mit dem Ausdruck *in front of* beschrieben.

Ähnliche Schwierigkeiten ergeben sich bei der Wahl des richtigen Adjektivs, um einer Aussage den nötigen Ausdruck zu verleihen. Ein deutsches Wort hat dabei oft mehrere Übersetzungen, die zwar alle richtig aber nur eines oder wenige wirklich geläufig sind. Hier stellt sich die Frage nach dem richtigen Synonym. Die folgenden Redewendungen sind äquivalent, aber welche wird am häufigsten verwendet?

It depends ...

- ... *largely* on the skills.
- ... *heavily* on the skills.
- ... *significantly* on the skills.
- ... *primarily* on the skills.
- ... *greatly* on the skills.

Unsicherheiten dieser Art haben erfahrenere Autoren bisher mit Anfragen an Internet-Suchmaschinen, wie *Google*¹, oder Online-Wörterbücher, wie *LEO*² und *dict.cc*³, zu lösen versucht. Die Möglichkeiten sind hierbei jedoch stark eingeschränkt, wenn es um die Suche und Bewertung ganzer Phrasen geht. Wörterbücher dienen in erster Linie dazu, einzelne Worte zu übersetzen. Sie bieten zwar inzwischen auch mögliche Verwendungsbeispiele des eingegebenen Wortes an, doch es ist nicht möglich speziell nach einer Kombination von Worten suchen zu lassen. Zudem fehlt jegliche Auskunft über die Geläufigkeit einer Phrase innerhalb der englischen Sprache.

Internet-Suchmaschinen können dagegen auch ganze Zeichenketten in Web-Dokumenten finden. Um die Fragestellung nach dem vorwiegend verwendeten Adjektiv aus dem letzten Beispiel zu beantworten, könnte eine Anfrage, die eine sogenannte Wildcard anstelle des gesuchten Wortes enthält, an Google gestellt werden. Der Screenshot 1.1 zeigt einen Ausschnitt der Google-Antwortseite für die entsprechende Anfrage „*depends * on the skills*“. Aus den Ergebnissen können zwar einige sinnvolle Adjektive, wie *largely*, *heavily*, *significantly* und *primarily* erfahren werden, doch es ist nicht ersichtlich, welches davon am häufigsten verwendet wird. Die Position eines Ergebnisses innerhalb dieser Liste, kann hierüber keine Aussage machen, da diese nach Dokumenten und nicht nach den Häufigkeiten der Phrasen sortiert ist. Als Konsequenz daraus könnte jede herausgefundene Variante noch einmal direkt angefragt und die Anzahl der gefundenen Dokumente ausgewertet werden. Dabei stellt allerdings die anschließende Beurteilung dieses Vergleiches ein weiteres Problem dar, da die Angabe über die Dokumentanzahl nur einer groben Schätzung entspricht.

¹ <http://www.google.com>

² <http://www.leo.org>

³ <http://www.dict.cc>

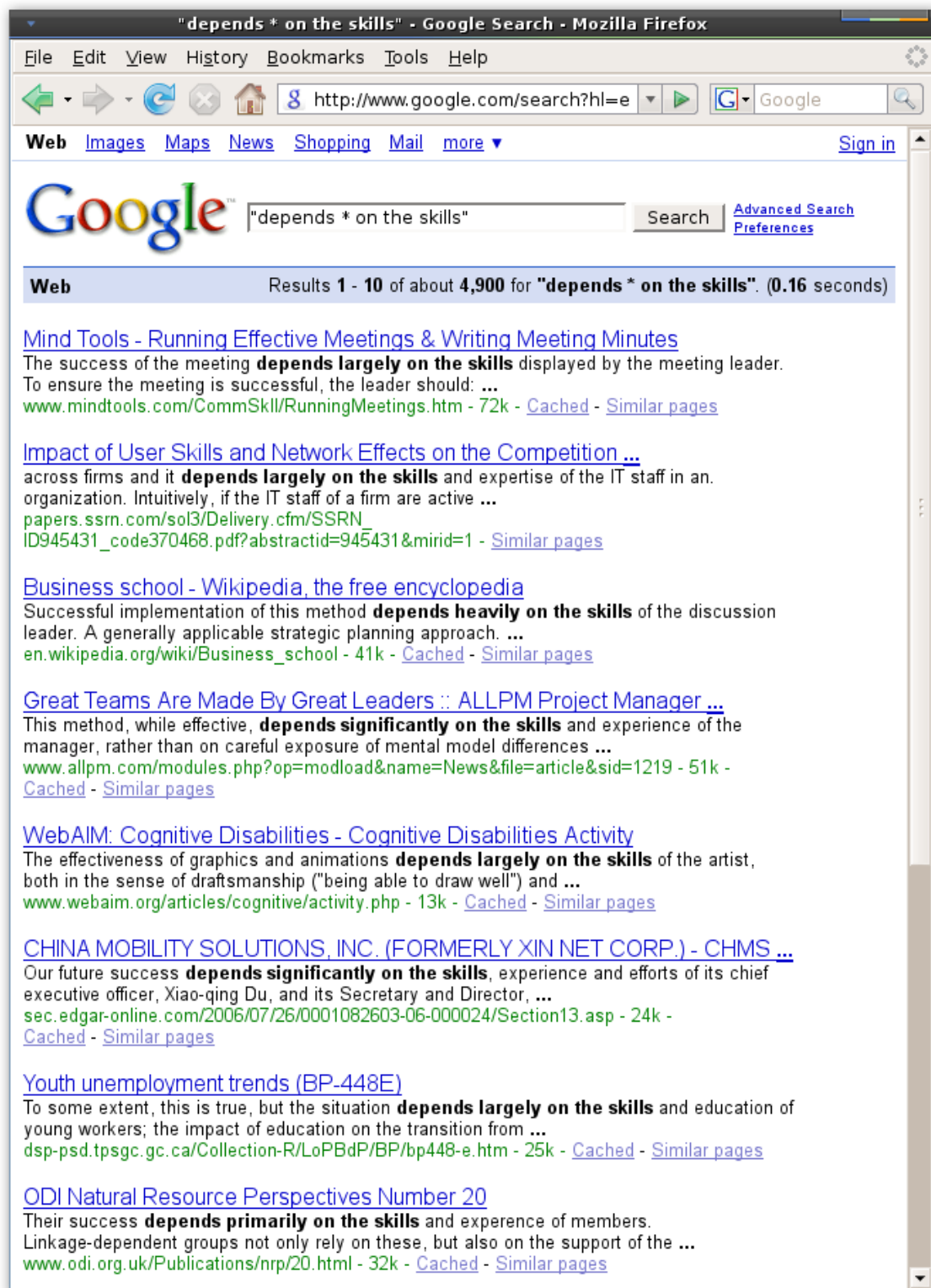


Abbildung 1.1: Google-Antwortseite nach der Anfrage „depends * on the skills“.

Es ist festzustellen, dass diese manuelle Analyse sehr zeitaufwändig und für umfangreiche Texte nahezu undurchführbar ist. Eine Automatisierung dieses Prozesses könnte daher viel Zeit sparen und die Qualität der Texte erheblich verbessern. Zu diesem Zweck wurde NETSPEAK entwickelt, ein Werkzeug, das Autoren bei der Aufgabe, fremdsprachige Texte zu verfassen, unterstützt.

1.1 Die Idee zu Netspeak

NETSPEAK ist ein neuartiges Web-Informationssystem, das es erlaubt, die Gebräuchlichkeit von kurzen Textphrasen in der englischen Sprache festzustellen. Als Wissensbasis dient dabei das World Wide Web. Das Web ist eine riesige Sammlung von Web-Dokumenten, die durch Millionen von Menschen erstellt wurde und täglich weiter wächst. Der Großteil dieser Dokumente ist in englischer Sprache von Muttersprachlern verfasst worden. Diese sogenannten Native-Speaker sind sehr sicher im Umgang mit ihrer Sprache und wissen, wie etwas richtig und zeitgemäß geschrieben wird. Über die Auswertung der Häufigkeit einer Formulierung im Web können also Rückschlüsse auf ihre Gebräuchlichkeit und Richtigkeit gezogen werden. Falsche oder unzeitgemäße Sprache wird darin weit weniger oft vorkommen als korrekte und geläufige Schreibweisen.

Da es viel zu aufwändig ist, das gesamte Web zu indizieren, um diese Idee zu realisieren, verwendet NETSPEAK eine Kollektion [IL06] (vgl. Tabelle 1.1) von kurzen Textphrasen, sogenannten N-Grammen, die auf Basis des gesamten Webs erstellt wurde. Ein N-Gramm ist in diesem Zusammenhang definiert als eine Kombination von N Worten. Die Erzeugung dieser N-Gramme wurde von Google, auf Grundlage aller indizierten englischsprachigen Web-Seiten, durchgeführt. Hierbei wurden sämtliche Markup-Informationen, wie HTML- und XML-Auszeichnungen, herausgefiltert und Wortfolgen bis zu einer Länge von fünf Worten protokolliert. Für jedes N-Gramm wurde zudem die Häufigkeit des Auftretens in den Web-Dokumenten bestimmt.

Die Google-N-Gramm-Kollektion umfasst zwar immer noch viel Speicherplatz, aber bedeutend weniger als alle englischsprachigen Dokumente des Web. Um Anfragen möglichst schnell beantworten zu können, ist eine effiziente Suche innerhalb dieser Kollektion notwendig. Es wurde daher ein spezieller Index nach dem Verfahren der invertierten Liste entwickelt, um die Menge der rund eine Milliarde 5-Gramme zu indizieren. Dieser spezialisierte Index liefert für ein gegebenes Wort, eine Liste von Referenzen auf 5-Gramme zurück, in denen dieses Wort vorkommt.

NETSPEAK stellt eine einfache Anfragesprache bereit, um in dieser Kollektion nach Phrasen suchen zu lassen. Dabei werden Platzhalter, sogenannte Wildcards, unterstützt, die durch sinnvolle Worte ersetzt werden. Das Ergebnis ist eine Liste von Vorschlägen gültiger Formulierungen, die anhand ihrer Häufigkeiten sortiert sind.

	Anzahl	Dateien	komprimierte Größe	unkomprimierte Größe
Token	1.024.908.267.229			
Sätze	95.119.665.584			
1-Gramme	13.588.391	1	70,2 Megabyte	177,00 Megabyte
2-Gramme	314.843.401	32	1,6 Gigabyte	5,0 Gigabyte
3-Gramme	977.069.902	98	5,5 Gigabyte	19,0 Gigabyte
4-Gramme	1.313.818.354	132	8,4 Gigabyte	30,5 Gigabyte
5-Gramme	1.176.470.663	118	8,8 Gigabyte	32,1 Gigabyte

Tabelle 1.1: Die Google N-Gramm-Kollektion.

Die Beiträge der vorliegenden Arbeit sind die Vorstellung sowohl des NETSPEAK-Dienstes als auch der Indizierungstechnologie, die auf den 5-Gramm-Korpus angewandt wurde. Da es sich bei NETSPEAK um ein Web-Informationssystem handelt - ein Begriff aus dem Information-Retrieval - führt Abschnitt 1.2 zunächst kurz die Grundbegriffe dieser Wissenschaft ein.

In Kapitel 2 wird die Funktionsweise von NETSPEAK beschrieben. Hierbei werden die Teilschritte des gesamten Retrieval-Prozesses im Detail erläutert, das Benutzer-Interface vorgestellt, und Wege diskutiert, wie die Performanz des prototypisch implementierten Systems weiter gesteigert werden kann.

Kapitel 3 gibt eine Einführung in die Indizierung großer Datenmengen. Es findet eine theoretische Beschreibung des Indizierungsverfahrens der invertierten Liste statt. Anschließend wird der speziell für NETSPEAK implementierte Index vorgestellt.

1.2 Information-Retrieval-Systeme

„Im Information-Retrieval (IR) werden Informationssysteme in Bezug auf ihre Rolle im Prozess des Wissenstransfers vom menschlichen Wissensproduzenten zum Informationsnachfragenden betrachtet“ [Fuh96]. Das heißt, Information-Retrieval ist eine inhaltsorientierte Suche und beschäftigt sich insbesondere mit der Semantik und Pragmatik von Dokumenten.

Das in Dokumenten formulierte Wissen wird dabei erst dann zu Information, wenn es von einem menschlichen Individuum in einer konkreten Situation zur Lösung eines Problems benötigt wird (vgl. [Kuh90]). Die Bereitstellung dieser Information und damit die Befriedigung des Informationsbedarfs ist Aufgabe eines IR-Systems.

Ein IR-System ist dabei oft ein komplexes verteiltes System, das zur Lösung einer Anfrage mehrere Wissensdatenbanken nutzt. Es ist dadurch gekennzeichnet, dass Anfragen vage

formuliert werden können und meist erst im Dialog interaktiv zu befriedigenden Antworten führen. Beispiele für IR-Systeme sind Börsen für Job- und Wohnungsangebote, Internet-Suchmaschinen oder Expertensysteme wie Routenplaner und Fahrkartenautomaten.

Die Form der Anfragesprache kann hierbei sehr unterschiedlich sein. Ideal wäre eine Formulierung in natürlicher Sprache, doch dazu müsste zuerst deren Semantik automatisiert verstanden werden können. Hiermit beschäftigen sich Methoden des Natural-Language-Processing, wobei viele Fragestellungen noch nicht gelöst sind.

Der am häufigsten anzutreffende Ansatz ist daher die Abstraktion einer Anfrage (Query) zu einem oder mehreren Schlüsselworten, zu denen passende Dokumente gefunden werden sollen. Sogenannte Boolean-Queries bieten die logischen Operatoren AND, OR und NOT an, um Schnittmengen, Vereinigungen und Differenzen von Dokumenten zu bilden. Andere Query-Formen erlauben eine exakte Mustersuche (Pattern-Matching) in Textdokumenten. Hierzu zählen auch Suchanfragen mit regulären Ausdrücken oder der Unterstützung von Wildcards. Wildcards sind bestimmte Sonderzeichen, die als Platzhalter für eines oder mehrere unbekannte Wörter stehen. Eine dritte Form bilden strukturierte Queries, die den Aufbau von Dokumenten berücksichtigen, wie er z.B. bei $\text{T}_{\text{E}}\text{X}$ -, HTML- und XML-Dokumenten gegeben ist. Hierbei kann speziell nach Autoren, Titeln oder Inhalten gesucht werden.

Eine zum Information-Retrieval verwandte Disziplin ist das Daten-Retrieval. Die Abgrenzung zum IR besteht darin, dass die Bereitstellung von Informationen nicht auf semantischer Ebene sondern auf syntaktischer bzw. sigmatischer Ebene stattfindet. Daher wird in diesem Zusammenhang eher von Daten als von Informationen gesprochen. DR-Systeme sollen Objekte aus einer Datenbank liefern, die exakt einer klar definierten Anfrage entsprechen. Anfragen haben dazu meist die Form von regulären Ausdrücken oder relationalen Algebren. Ein weiteres Merkmal ist das Fehlen von Ranking-Mechanismen, um Ergebnisse nach bestimmten Relevanzkriterien zu bewerten und zu sortieren. Beispiele für DR-Systeme sind Datenbank-Management-Systeme mit ihren dazugehörigen Anfragesprachen (z.B. *MySQL*⁴). DR-Systeme können auch Subkomponenten eines größeren IR-Systems bilden. Eine detaillierte Unterscheidung wird in [vR79] gegeben.

Der schematische Aufbau eines IR-Systems ist in Abbildung 1.2 vereinfacht dargestellt (vgl. [BYRN99]). Zu Beginn eines IR-Prozesses formuliert der Benutzer seinen Informationsbedarf in Form der jeweiligen Anfragesprache und sendet diesen über ein Benutzer-Interface an das System. Die Query wird hier bestimmten Textoperationen unterzogen, um sie in ein internes Dokumentmodell abzubilden (z.B. das Vektorraummodell). Diese abstrakte Repräsentation dient der Suche nach passenden oder ähnlichen Dokumenten, was auch als Query-Expansion bezeichnet wird. In der Regel findet dabei u.a. ein Index-Lookup statt, der Referenzen auf passende Dokumente liefert. Mit diesen Referenzen können die

⁴ <http://www.mysql.com>

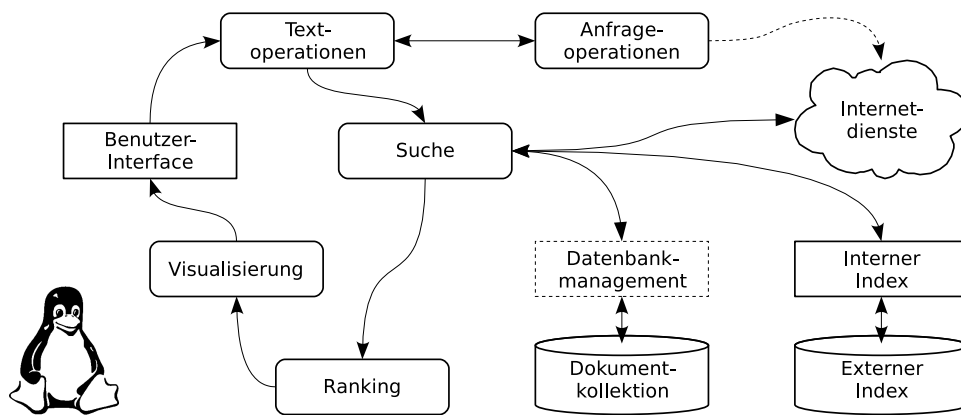


Abbildung 1.2: Allgemeiner Information-Retrieval-Prozess.

eigentlichen Dokumente direkt oder indirekt, über ein Datenbankmanagement, aus einer Kollektion ausgelesen werden. Anschließend folgt ein Ranking der ermittelten Dokumente, um diese nach bestimmten Relevanzkriterien zu sortieren. Dieser Schritt hat großen Einfluss auf die Qualität der Suche. Schließlich wird das Ergebnis des Retrieval-Prozesses im Benutzer-Interface visualisiert.

Bei dem angesprochenen Index handelt es sich um eine verteilte Datenstruktur, die im Vorfeld aus der Dokumentkollektion generiert wurde, um diese schnell durchsuchen zu können. Bekannte, frei verfügbare Indizierungsbibliotheken sind *Lucene*⁵ und *Terrier*⁶. Für NETSPEAK wurde eine eigene Implementierung entwickelt.

⁵ <http://lucene.apache.org>

⁶ <http://ir.dcs.gla.ac.uk/terrier>

2 Der Web-Dienst Netspeak

NETSPEAK ist als klassischer Web-Dienst implementiert, der von einem Benutzer über einen grafischen Internet-Browser bedient werden kann. Die Startseite von NETSPEAK kann unter der Adresse <http://netspeak.webis.de> aufgerufen werden. Sie ist als Screenshot in Abbildung 2.1 dargestellt.

Die Oberfläche lässt sich zunächst in zwei Abschnitte unterteilen: Im oberen Teil befindet sich rechts neben dem NETSPEAK-Logo die Suchmaske. Diese besteht aus einem Eingabefeld für die Query und einer Schaltfläche zum Absenden. Darunter folgt eine Anzeige mit Beispielen gültiger Queries, die bereits einen Einblick in die Syntax der Anfragesprache geben. Nach einer Anfrage erscheint ein dritter Abschnitt, in dem die Suchergebnisse präsentiert werden.



Abbildung 2.1: Die Startseite von NETSPEAK.

2.1 Der Retrieval-Prozess

Die Bearbeitung einer Anfrage an NETSPEAK entspricht einem typischen IR-Prozess, so wie er in Abschnitt 1.2 erläutert wurde. Abbildung 2.2 zeigt diesen Prozess speziell für NETSPEAK als UML-Aktivität.

Über das Benutzer-Interface wird eine Query eingegeben und an das System übermittelt. Der erste Schritt besteht darin, dass die Syntax der Query überprüft wird. Diese muss einer bestimmten Grammatik entsprechen, die in Abschnitt 2.2.1 definiert wird. Handelt es sich dabei um eine ungültige Anfrage, wird die Bearbeitung abgebrochen und ein leeres Ergebnis zurückgeliefert.

Im anderen Fall folgt als zweiter Schritt die Spell-Correction. Hierbei wird eine Anfrage auf Rechtschreibfehler hin untersucht und mögliche korrigierte Queries erzeugt, die dem Benutzer auf der Antwortseite vorgeschlagen werden.

Parallel dazu findet die Query-Expansion statt, bei der alle N-Gramme ermittelt werden, die auf die eingegebene Anfrage zutreffen. Für entsprechend gekennzeichnete Worte der Query werden hierbei zunächst Synonyme gesucht mit denen weitere Queries generiert werden. Für jedes Wort einer Query findet daraufhin ein Lookup im N-Gramm-Index statt. Ein Lookup liefert dabei eine Liste von Referenzobjekten zurück. Diese Objekte verweisen auf die Menge derjenigen N-Gramme im N-Gramm-Korpus, in denen das betrachtete Wort vorkommt. Ein Referenzobjekt enthält zudem die genaue Position dieses Wortes innerhalb des adressierten N-Gramms.

Aus allen gelieferten Referenzen wird die Schnittmenge derjenigen Referenzen gebildet, die auf N-Gramme verweisen, die alle Worte aus der Query in der richtigen Reihenfolge enthalten. Dieser Prozess wird zusammenfassend als Mustersuche im N-Gramm-Index bezeichnet.

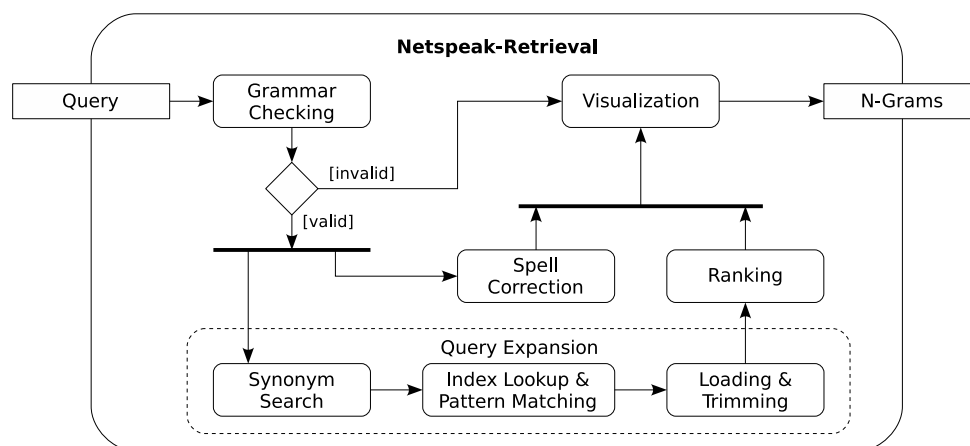


Abbildung 2.2: NETSPEAK-Retrieval-Prozess als UML-Aktivität.

Die durch diese Schnittmenge referenzierten N-Gramme werden schließlich aus dem N-Gramm-Korpus eingelesen. Es wird das Muster, das durch die Query definiert wird, innerhalb dieser N-Gramme identifiziert. Die N-Gramme werden gegebenenfalls beschnitten und gleiche Übereinstimmungen zusammengefasst. Nicht konforme N-Gramme werden herausgefiltert.

Anschließend wird das Ranking der verbliebenen N-Gramme anhand ihrer Häufigkeiten durchgeführt. Das Ergebnis wird zuletzt an den Client gesandt, wo die Visualisierung stattfindet.

2.2 Syntax der Anfragesprache

Die Anfragesprache von NETSPEAK ist der Klasse von Queries zuzuordnen, die auf einer wortorientierten Mustersuche basieren. Im Gegensatz zu Boolean-Queries, die anhand von Schlüsselworten eine inhaltliche Suche nach Dokumenten durchführen, wird hierbei nach einer exakter Übereinstimmung von Textphrasen gesucht und entsprechende Dokumente zurückgeliefert. Im Falle von NETSPEAK handelt es sich dabei um eine Menge von N-Grammen.

Es werden zunächst zwei Grammatiken unterschieden: Die allgemeine und die spezielle Query-Grammatik. Die allgemeine Grammatik definiert den Aufbau einer gültigen Anfrage, die ein Benutzer über das Web-Interface eingeben kann. Eine diesbezügliche Überprüfung findet unter Grammar-Checking (vgl. Abb. 2.2) statt. Aus der Instanz einer eingegebenen gültigen Query wird dann eine spezielle Grammatik erzeugt, die für die Mustersuche im N-Gramm-Index sowie für das Beschneiden (Trimmen) und Zusammenfassen von N-Grammen verwendet wird.

2.2.1 Allgemeine Query-Grammatik

Diese Grammatik beschreibt die Regeln der Anfragesprache an den NETSPEAK-Dienst, die anhand eines einführenden Beispiels vorgestellt wird. Die Anfrage eines Benutzers lautet:

```
depends * ~skill
```

Die Query besteht aus den beiden Worten `depends` und `skill`, die durch einen Stern getrennt sind. Das zweite Wort ist mit einer vorangestellten Tilde gekennzeichnet. Der Stern drückt aus, dass an dieser Stelle sinnvolle Worte ergänzt werden sollen und die Tilde fragt nach möglichen Synonymen für `skill`. Der Benutzer sucht also geläufige Formulierungen, in denen diese Worte genau in der Reihenfolge, aber mit variablen Abstand, auftauchen und gegebenenfalls ein Synonym für `skill` enthalten, das womöglich gebräuchlicher ist.

```

QUERY    = { WORD | SYNWORD | WILDCARD } ;
WORD     = ( [ APOS ] ( LETTER { LETTER } ) ) | KOMMA ;
LETTER   = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
          "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
          "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
          "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
          "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
          "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
SYNWORD  = TILDE WORD ;
WILDCARD = "?" | "*" ;
TILDE    = "~" ;
KOMMA    = "," ;
APOS     = "'" ;

```

Abbildung 2.3: Die Grammatik der Anfragesprache in EBNF.

Die genaue Query-Grammatik ist in Abbildung 2.3 in EBNF-Notation (Erweiterte-Backus-Naur-Form) dargestellt. Eine gültige Anfrage besteht demnach aus einer beliebigen Kombination von Worten, Synonymworten und Wildcards. Ein Wort ist - bis auf zwei Spezialfälle, die später erläutert werden - als eine beliebige Buchstabensequenz definiert. Eine Tilde zusammen mit einem Wort ergibt ein Synonymwort. Das Fragezeichen und der Stern bilden jeweils eine Wildcard.

Eine sehr spezielle Anfrage besteht in der Eingabe einer beliebigen Wortkombination, um deren Existenz im Web und damit ihre Richtigkeit zu überprüfen. Für allgemeine Anfragen werden hingegen die Wildcards eingesetzt. Sie dienen als Platzhalter, die später im Ergebnis durch sinnvolle Worte ersetzt werden. Der Tilde-Operator markiert ein gegebenes Wort für eine Synonymsuche. Die Semantik der beiden Wildcards orientiert sich an den regulären Ausdrücken und ist wie folgt definiert:

- Das Fragezeichen (?) steht für genau ein Wort.
- Der Stern (*) steht für kein bis beliebig viele Worte.

Fragezeichen können mehrfach hintereinander auftreten, um genau zwei, drei oder vier Worte ergänzen zu lassen. Eine Kombination von Wildcards mit mindestens einem Stern evaluiert dabei zu einem Stern. Leerzeichen zwischen Wildcards bzw. zwischen einer Wildcard und einem Wort sind optional. Die Tilde und das dazugehörige Wort dürfen nicht getrennt werden. Folgende Queries sind somit äquivalent:

- WORD ? ? ~WORD
- WORD ?? ~WORD
- WORD??~WORD

Eine besondere Rolle in der Anfragesprache spielen die beiden Interpunktionen Komma und Apostroph. Diese müssen gesondert behandelt werden, wenn später bei der Muster-

suche im N-Gramm-Index anhand der Wortabstände die Menge passender N-Gramme ermittelt wird. Während das Komma innerhalb eines N-Gramms aus dem N-Gramm-Korpus als einzelnes, durch Leerzeichen getrenntes, Token auftritt, ist das Apostroph Bestandteil der sich rechtsseitig anschließenden Wortverkürzung. Ein Beispiel dazu wäre `without it , you 're`, wobei das Komma und `'re` jeweils eigene Token repräsentieren.

Diese Situation wird bereits beim Tokenisieren der Query durch Anwendung der allgemeinen Query-Grammatik berücksichtigt. Hierbei wird das Komma als ein Wort und die durch ein Apostroph getrennte Zeichenketten als zwei Worte gezählt.

Für das Query-Parsing wird eine sehr leistungsfähige Bibliothek namens *ANTLR*¹ (Another Tool for Language Recognition) verwendet. Dieser Parser-Generator erlaubt die Definition einer komplexen Grammatik über eine Textdatei in EBNF-Notation, aus der anschließend eine Lexer- und eine Parser-Klasse in Java-Quellcode erstellt werden. Der Lexer erzeugt zunächst aus einem übergebenen Query-String einen Tokenstrom, der dann an den Parser weitergeleitet wird. Dieser Parser entspricht einem endlichen Automaten, für den in jedem Zustand benutzerdefinierter Code ausgeführt werden kann, um auf bestimmte Ereignisse zu reagieren. Diese Möglichkeit wird genutzt, um während des Parsings ein internes Query-Objekt zu erzeugen.

2.2.2 Spezielle Query-Grammatik

Während die allgemeine Query-Grammatik eine gültige Anfrage als eine Kombination von Worten und Wildcards definiert, wird durch eine eingegebene Query-Instanz eine spezielle Grammatik festgelegt. Ein Wort der Query wird dabei von einem Nicht-Terminalsymbol zu einem Terminalsymbol. Die folgenden zwei Instanzen verdeutlichen noch einmal den Abstraktionsgrad der beiden Grammatiken.

- Eine Query nach der allgemeinen Grammatik: `WORD ?? WORD`
- Durch Query induzierte spezielle Grammatik: `depends ?? skill`

Eine spezielle Grammatik wird für die Mustersuche im N-Gramm-Index verwendet, um die Menge von N-Grammen zu bestimmen, die mit dem Muster der eingegebenen Query übereinstimmen. Ist dabei die Query kürzer als die N-Gramme, ist es sehr wahrscheinlich, dass zwei unterschiedliche N-Gramme die Query auf dieselbe Art und Weise erfüllen. Die Query `depends ?? skill` liefert u.a.:

- `it depends on your skill`
- `depends on your skill and`

¹ <http://www.antlr.org>

Für die Aufbereitung der Retrieval-Ergebnisse müssen die N-Gramme beschnitten und solche Duplikate zusammengefasst werden. Dazu ist eine erneute exakte Mustersuche notwendig, die in Abschnitt 2.4.3 beschrieben wird.

2.3 Spell-Correction

Der Prozess der Spell-Correction untersucht die Worte einer Anfrage auf Rechtschreibfehler und erzeugt mögliche Korrekturen. Diese korrigierten Queries gehen nicht in die Ergebnissuche mit ein, sondern werden auf der Antwortseite unter dem Stichwort *Did you mean* angezeigt.

Für die Rechtschreibkorrektur wird eine Implementation des *SmartSpell*-Algorithmus verwendet [SC06]. Die Überprüfung von Worten geschieht hierbei mittels Identifizierung einzelner Phoneme, die zur Erkennung des gemeinten Wortes benutzt werden. Als Phoneme werden Laute einer gesprochenen Sprache bezeichnet, die eine bedeutungsunterscheidende Funktion haben. Die ermittelten Vorschläge werden zudem anhand eines Ähnlichkeitsmaßes gewichtet. Dieser Wert wird in Prozent angegeben. Abbildung 2.4 zeigt einige Beispiele für eine Korrektur mittels SmartSpell.

Liefert SmartSpell für mehrere Worte einer Query auch mehrere Korrekturvorschläge, so kann aus jeder möglichen Kombination eine neue Query generiert werden. Angenommen es gäbe drei Worte mit jeweils drei Korrekturen, dann entspräche das 27 Möglichkeiten. Um dem Benutzer die wahrscheinlichste Korrektur seiner Query zuerst zu präsentieren, werden die möglichen Varianten gewichtet. Dazu werden die Einzelgewichte der ersetzten Worte innerhalb einer korrigierten Query zu einem Gesamtgewicht aufsummiert. Die drei Queries mit dem höchsten Gewicht werden als Korrekturvorschlag ausgewählt.

Ein Problem der verwendeten SmartSpell-Implementation ist die fehlende Unterstützung einer exakten Korrektur für flektierte Wortformen. So bleiben Nachsilben, wie sie bei Pluralformen von Substantiven und der dritten Person Singular von Verben auftreten, in den vorgeschlagenen Worten nicht erhalten. Für ein richtig gebeugtes Wort wird sogar die Grundform als Korrektur zurückgeliefert. Es ist somit schwierig festzustellen, ob tatsächlich ein brauchbarer Vorschlag vorliegt. Ursache hierfür ist nicht der Algorithmus, sondern die Verwendung eines zu kleinen Vokabulars zur Korrektursuche. Ein umfangreicheres Wörterbuch, das alle möglichen Flexionen enthält, könnte das Problem auf Dauer lösen. NETSPEAK versucht diese Situation zu klären, indem es einen Test durchführt, ob es sich bei einer Korrektur um einen Substring des ursprünglichen Wortes handelt. Trifft dies zu, so wird die vermeintliche Korrektur abgelehnt.

angeneering	→	engineering 92%
buysikel	→	physical 85%, bicycle 82%
kompilayshon	→	compilation 89%

Abbildung 2.4: Rechtschreibkorrekturen mittels SmartSpell.

2.4 Query-Expansion

Der Prozess der Query-Expansion entspricht bei NETSPEAK der Suche nach allen N-Grammen, die auf eine (Wildcard-) Query matchen. Hierfür sind drei Teilschritte zu unterscheiden: 1. Die optionale Suche nach Synonymen zur Erzeugung weiterer Queries, 2. Die Mustersuche im N-Gramm-Index, um die Menge der N-Gramme zu ermitteln, die die Query erfüllen, und 3. Eine erneute exakte Mustersuche, um Duplikate zu identifizieren und zusammenzufassen.

2.4.1 Synonymsuche

Die optionale Suche nach Synonymen hat innerhalb der Query-Expansion den Zweck festzustellen, welches der möglichen Synonyme am häufigsten verwendet wird. Bei NETSPEAK wird diese für diejenigen Worte einer Query durchgeführt, die mit einer vorangestellten Tilde gekennzeichnet sind. Für die Synonyme eines Wortes werden dann automatisch weitere Query-Instanzen erzeugt. Werden hierbei mehrere Worte auf diese Weise markiert, werden nur für die ersten zehn möglichen Kombinationen von Synonymen Queries generiert, um die Antwortzeit akzeptabel zu halten. Ein Beispiel für die Expansion der Query `depends * ~skill` lautet:

- `depends * skill`
- `depends * accomplishment`
- `depends * acquirement`
- `depends * acquisition`
- `depends * attainment`

Zur Realisierung dieser Funktion wird eine Bibliothek namens *JAWS*² (Java API for WordNet Searching) verwendet. JAWS ist eine API zum Zugriff auf *WordNet*³, einem Online-Synonymlexikon der englischen Sprache für Substantive, Verben, Adjektive und Adverbien. Da WordNet seine Datenbank zum Herunterladen anbietet, arbeitet NETSPEAK mit einer lokalen Kopie, um die Suche zu beschleunigen.

² <http://enr.smu.edu/~tspell>

³ <http://wordnet.princeton.edu>

Eine Anfrage an WordNet mittels JAWS findet unter Angabe eines Wortes und dessen Wortart statt. Letzteres wird zuvor mit Hilfe der Bibliothek *qtag* bestimmt. Das Ergebnis ist eine Menge von sogenannten Synsets. Ein Synset repräsentiert dabei Synonyme mit gleicher Bedeutung. Das Adverb *before* liefert z.B. die folgenden beiden Synsets:

- *earlier, before*
- *ahead, in front, before*

Das erste Synset enthält Synonyme für *before* im Sinne einer zeitlichen Relation, während das zweite eine physikalische Gegebenheit beschreibt. Es stellen sich nun zwei Fragen: 1. Welches Synset enthält diejenigen Synonyme, die für den gegebenen Kontext die richtige Semantik wiedergeben und 2. welches Synonym, aus dem richtigen Synset, am häufigsten verwendet wird. Da diese Fragestellung genau dem Informationsbedarf des Benutzers entspricht, sucht NETSPEAK nach Phrasen für alle Synonyme.

Ein Problem stellt dagegen, ähnlich wie bei der Rechtschreibkorrektur, die Synonymsuche für flektierte Worte dar. WordNet erkennt zwar gebeugte Wortformen, liefert aber deren Synonyme in der Grundform zurück. Da diese Synonyme ohne Wiederherstellung der korrekten Form in die Query eingefügt werden, kann das gewünschte Ergebnis, weitere sinnvolle N-Gramme zu finden, gegebenenfalls nicht erreicht werden. Die Lösung hierfür wäre eine automatische Feststellung der Flexion mit anschließender Rückführung. Eine entsprechende Datenbank, allerdings für die deutsche Sprache, liefert z.B. *canoonet*⁴.

2.4.2 Mustersuche im N-Gramm-Index

Die Suche im N-Gramm-Index dient der Bestimmung der Menge aller N-Gramme, die auf das Muster, das von einer Query vorgegeben ist, passen. Das heißt, dass diese N-Gramme alle Worte der Query in der richtigen Reihenfolge unter Berücksichtigung der Semantik auftretender Wildcards beinhalten müssen. Der N-Gramm-Index bietet dazu die Möglichkeit, für ein bestimmtes Wort eine Liste mit Referenzen anzufragen, die auf eine Menge von N-Grammen im N-Gramm-Korpus verweist, in denen dieses Wort vorkommt. Für eine gegebene Query werden die Referenzlisten der enthaltenen Worte aus dem Index ausgelesen, um daraus eine Schnittmenge zu bilden, die nur noch diejenigen Referenzen enthält, die die gesuchten N-Gramme adressieren.

Um das Vorgehen zur Bildung dieser Schnittmenge im Detail zu erläutern, wird zunächst der Aufbau dieses Referenzobjekts vorgestellt. Dabei handelt es sich um ein 5-Tupel, bestehend aus einer Datei-Id, einem Offset, einer Bytelänge, einem Häufigkeitswert und einer Positionsangabe. Die ersten beiden Elemente dieses 5-Tupels adressieren eindeutig ein N-Gramm aus dem N-Gramm-Korpus und die Positionsangabe verweist auf genau ein Wort

⁴ <http://www.canoo.net>

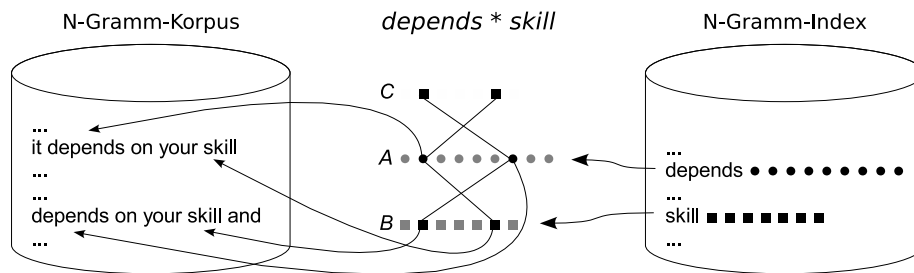


Abbildung 2.5: Mustersuche im N-Gramm-Index.

innerhalb dieses N-Gramms. Eine Instanz dieses Objektes verweist also sowohl auf ein N-Gramm als auch auf ein bestimmtes Wort darin. Es gilt, dass zwei Instanzen ein gemeinsames N-Gramm referenzieren, wenn sie in Datei-Id und Offset übereinstimmen. Der Abstand der adressierten Worte kann über die Differenz der Wortpositionen bestimmt werden.

Abbildung 2.5 demonstriert die Schnittmengenbildung am Beispiel der Query `depends * skill`, die nach dem Prozess des Grammar-Checking (vgl. 2.2) als Tokenstrom vorliegt. Für das erste Wort `depends` wird die dazugehörige Referenzliste *A* aus dem Index angefragt, die zunächst die Ergebnisliste bzw. Schnittmenge bildet. Beim zweiten Token, dem Stern, handelt es sich um eine Wildcard. Der durch diese Wildcard repräsentierte Wortabstand *d* wird in einer Variable zwischengespeichert. Als nächstes folgt das Wort `skill`, deren Referenzliste *B* wiederum aus dem Index ausgelesen wird. Diese Referenzliste wird mit der aktuellen Ergebnisliste *A*, unter dem einzuhaltenden Wortabstand *d*, geschnitten. Das Resultat dieses Schnittes ist die Referenzliste *C*, die die neue Ergebnismenge bildet. *C* verweist auf die beiden N-Gramme `it depends on your skill` und `depends on your skill and`.

Aus einer Verkettung von Einzelschnitten zweier Referenzlisten ergibt sich also die endgültige Schnittmenge, die sich dabei sukzessive verkleinert. Für einen Einzelschnitt zwischen zwei Referenzlisten *B* und *A* unter dem Wortabstand *d* sind allgemein zwei Kriterien zu überprüfen: 1. Ein $b \in B$ ist Element der resultierenden Schnittmenge *C*, wenn auch ein $a \in A$ existiert, das auf dasselbe N-Gramm verweist, und 2. die Differenz der beiden Wortpositionen in *a* und *b* mit *d* übereinstimmt. 1. kann hierbei performant über Hashtabellen festgestellt werden und 2. erfordert eine Subtraktion. Für zwei direkt aufeinander folgende Worte gilt dabei immer $d = 1$ und beim Auftreten der Wildcard `?` entsprechend $d = 2$. Der Stern impliziert $d \geq 1$. Es ist zu beachten, dass die Elemente der Ergebnismenge *C* als Wortposition den Wert der Elemente aus *B* annehmen, um die oben beschriebene Verkettung von Schnitten zu ermöglichen.

Es kann vorkommen, dass zu einem gegebenen Wort keine Referenzliste im N-Gramm-Index existiert. In diesem Fall wird der aktuelle Wortabstand *d* für den nächsten stattdin-

denden Schnitt um eins erhöht. Gleiches gilt für das Komma und solche Token, die mit einem Apostroph beginnen. Ein exakter Musterabgleich erfolgt dann beim Trimmen und Zusammenfassen der N-Gramme.

2.4.3 Zusammenfassen von N-Grammen

Aus dem vorherigen Prozess der Mustersuche im N-Gramm-Index stehen nun diejenigen Referenzen zur Verfügung, die auf die gesuchten N-Gramme verweisen. Dabei ist es allerdings sehr wahrscheinlich, dass sich darunter mehrere Instanzen mit gleicher Übereinstimmung befinden. Dieser Fall tritt ein, wenn die Expansion der Query kürzer ist als die N-Gramme. Die Query `depends * skill` liefert u.a.:

- `it depends on your skill`
- `depends on your skill and`

Da am Anfang sowie am Ende der Query keine Wildcards gesetzt sind, müssen `it` und `and` entfernt werden, was in diesem Zusammenhang als Trimming bezeichnet wird. Die verbleibenden N-Gramme bilden Duplikate und werden vereinigt.

Diese Prozedur wird auf die maximal 100 ersten N-Gramme der Schnittmenge angewandt, die dafür zunächst aus dem N-Gramm-Korpus eingelesen werden. Diese Teilmenge entspricht der Menge mit den größten Häufigkeitswerten, da alle Referenzlisten und somit auch die Schnittmenge bereits absteigend danach sortiert vorliegen. Was folgt ist eine erneute exakte Mustersuche anhand der speziellen Query-Grammatik mit Hilfe eines regulären Ausdrucks. Dieser muss hierfür nur einmalig für die gegebene Query kompiliert werden und liefert bei einer erfolgreichen Suche den genauen Anfang und das Ende des matchenden Substrings innerhalb des N-Gramms zurück. Dieser Substring wird extrahiert und in die Datenstruktur eines Sets eingefügt, die dabei automatisch Duplikate zusammenfasst. Der reguläre Ausdruck für oben gezeigtes Beispiel, wobei `.*` für eine beliebige Folge von Buchstaben steht, lautet: `depends .* skill`

Beim Zusammenfassen von Duplikaten werden zudem die Häufigkeitswerte der beteiligten N-Gramme aufsummiert und dem getrimmten N-Gramm zugewiesen. Hierbei ist es möglich, dass es sich bei zwei oder mehreren Duplikaten um dieselben Instanzen handelt. Das obige Beispiel könnte etwa aus ein und demselben Satz stammen. Der Häufigkeitswert des N-Gramms ist also nach oben hin verfälscht. Da dieser Effekt allerdings bei jedem resultierenden N-Gramm auftritt, bleiben die Ergebnisse relativ zueinander vergleichbar.

Eine weitere Funktion dieser exakten Mustersuche ist die Herausfilterung von N-Grammen, die zwar durch die Mustersuche im N-Gramm-Index ermittelt wurden, aber tatsächlich nicht zur Query passen. Dieser Fall kann eintreten, wenn für ein bestimmtes Wort keine Referenzliste im N-Gramm-Index zur Verfügung stand. Ein Beispiel einer Query für diese Situation lautet: `without it , you 're`

Zu den beiden Token `,` und `'re` gibt es keine Referenzlisten im Index. Die Berechnung der Schnittmenge basiert aber genau auf der Auswertung dieser Referenzen, um anhand der Wortabstände passende N-Gramme zu bestimmen. Solche Worte werden dann wie die Wildcard `?` behandelt. Für die Mustersuche im N-Gramm-Index ist also diese Query äquivalent zu: `without it ? you ?`

In der Schnittmenge befinden sich also Referenzen bzw. N-Gramme, die zwar mit dem Muster `without it ? you ?` übereinstimmen, aber nicht mit der eigentlichen Query `without it , you 're`. Bei Anwendung der speziellen Grammatik durch den regulären Ausdruck werden solche N-Gramme identifiziert und nachträglich herausgefiltert.

2.5 Ranking nach Häufigkeiten

Das Ranking bezeichnet die Gewichtung der gefundenen N-Gramme hinsichtlich ihrer Relevanz in Bezug auf die eingegebene Query. Hierfür werden die Häufigkeiten der N-Gramme und der einzelnen Worte ausgewertet. Dabei wird angenommen, dass der Häufigkeitswert als Indikator für deren Richtigkeit oder Gebräuchlichkeit dient. Ein höherer Wert wird demnach als relevanter betrachtet. Es werden ein absolutes Ranking und mehrere relative Rankings unterschieden.

Das absolute Ranking orientiert sich an den absoluten Häufigkeitswerten der N-Gramme, um deren Verbreitung im World Wide Web darzustellen. Die Ergebnismenge der N-Gramme wird dementsprechend in absteigender Reihenfolge sortiert. Das erste Element dieser Menge repräsentiert somit das am häufigsten im Web vorkommende N-Gramm.

Bei den relativen Rankings hingegen werden die N-Gramme nach ihrer Häufigkeit in Relation zur Häufigkeit eines Wortes aus der Query gerankt: Die absolute Häufigkeit des N-Gramms wird geteilt durch die absolute Häufigkeit des betrachtete Wortes. Solche Rankings werden jeweils für alle Worte aus der Query durchgeführt.

2.6 Visualisierung im Web-Interface

Die Visualisierung des Retrieval-Ergebnisses findet im Internet-Browser des Benutzers statt. Alle ermittelten N-Gramme werden hierfür, zusammen mit den berechneten absoluten und relativen Häufigkeiten, an den Client gesandt. Eine Antwortseite für die Query `depends * ~attainment` zeigt der Screenshot [2.6](#).

Netspeak

Type one or more word or ~synword or ? or *

Ranking	Results
absolute	depends * skill
8231	depends on the skill
2912	depends on your skill
1289	depends on skill
616	depends upon the skill
185	depends upon your skill
144	depends largely on the skill
109	depends on strategy and skill
88	depends heavily on the skill
82	depends entirely on the skill
67	depends more on the skill
60	depends greatly on the skill
50	depends on retooling your skill
46	depends on user skill
44	depends largely upon the skill
41	depends on practice and skill
40	depends somewhat on the skill
absolute	depends * acquisition
245	depends on the acquisition
88	depends upon the acquisition
80	depends on the particular acquisition
47	depends both on preventing acquisition
absolute	depends * attainment
106	depends on the attainment
65	depends on the educational attainment
absolute	depends * accomplishment
40	depends on the accomplishment
absolute	depends * acquirement

© 2008 [Bauhaus University Weimar](#) | [Webis](#) | [Help](#)

Abbildung 2.6: Eine Antwortseite von NETSPEAK.

Die Darstellung der Ergebnisse findet in Form einer zweiseitigen Tabelle statt. Jede Zeile zeigt ein N-Gramm, das zur Query passt sowie seine absolute oder relative Häufigkeit. Die Zeilen sind nach den Häufigkeiten sortiert. Ergebnisse automatisch generierter Queries durch die Synonymsuche werden abschnittsweise dargestellt. Das Ranking der N-Gramme kann über ein Drop-Down-Menü im Kopf der Tabelle bzw. des Abschnitts umgeschaltet werden. Das Menü enthält dabei die Einträge „*absolute*“ sowie eine Liste der einzelnen Worte der Query.

Bei Auswahl eines anderen Rankings wird die Tabelle entsprechend neu aufgebaut. Hierbei findet keine weitere Kommunikation mit dem NETSPEAK-Server statt. Alle benötigten Daten sind bereits in die Antwortseite, in Form von JavaScript-Objekten, eingebettet. Die Änderung der Anzeige geschieht also durch eine dynamische Modifikation des HTML-Dokuments auf dem Client.

Abbildung 2.7 zeigt einen weiteren Screenshot einer Antwortseite zur Demonstration der Rechtschreibkorrektur. In der Query *intristed ? compjuter sciencz* sind alle Wörter falsch geschrieben. Es werden demnach keine passenden N-Gramme gefunden. Unter dem Stichwort *Did you mean* werden die Vorschläge zu möglichen Korrekturen aus dem Prozess der Spell-Correction dargestellt. Der erste Eintrag *interested ? computer science* entspricht der korrekten Query. Diese kann direkt angeklickt werden, um danach zu suchen.



Abbildung 2.7: Eine Antwortseite mit Vorschlägen zur Rechtschreibkorrektur.

2.6.1 Nachladen von Textbeispielen

Das GUI soll die zusätzliche Funktion anbieten, Textbeispiele für jedes einzelne N-Gramm zu laden, in denen es verwendet wird. Es ist geplant in der Ergebnistabelle rechts neben jedem N-Gramm ein klickbares Icon zu positionieren, über welches das Nachladen gestartet werden kann. Um nicht den NETSPEAK-Server mit dieser Aufgabe unnötig zu belasten, werden hierbei Internet-Suchmaschinen direkt angesprochen. Der empfangene Text wird aufbereitet und dynamisch in die aktuelle Ergebnisseite eingefügt.

Die notwendige Technologie zum asynchronen Senden und Empfangen dieser HTTP-Anfragen wird durch AJAX zur Verfügung gestellt. In Verbindung mit einer API, wie z.B. *Google AJAX Search*⁵, werden die Internet-Suchmaschinen angefragt.

2.7 Performanz des Web-Dienstes

Dieser Abschnitt diskutiert Aspekte, die die Performanz von NETSPEAK beeinflussen. Da bei einem Web-Dienst geringe Antwortzeiten von großer Bedeutung sind, ist eine durchweg effiziente Implementierung sehr wichtig. Ein Dienst, auf dessen Antworten ein Benutzer lange warten muss, ist schnell nicht mehr die erste Wahl, wenn es Alternativen mit gleicher Qualität gibt. Hierauf ist letztendlich auch der durchschlagende Erfolg von Google zurückzuführen. Ein Flaschenhals in der Anfragebearbeitung kann wichtige Zeit kosten, in der ein Benutzer sich möglicherweise vom Dienst abwendet.

Ein performanter Dienst beginnt bereits mit der richtigen Wahl der Hardware und des Betriebssystems und endet bei der verwendeten Programmiersprache und den Algorithmen. Wo das meiste Potenzial zur Beschleunigung liegt, sollte auch der größte Aufwand betrieben werden sie zu realisieren. Ein schnellerer Algorithmus bringt keinen Vorteil, wenn etwa die Festplatte nicht schnell genug die benötigten Daten bereitstellen kann. Eine wichtige Rolle spielt hierbei die Parallelisierung von Prozessen. Dabei gilt es herauszufinden, welche Teile eines Programms seriell ausgeführt werden müssen und welche parallelisiert werden können. Nach dem Gesetz von Amdahl [Wik08a] wird hierbei der zu erreichende Speedup mit steigender Parallelisierung immer mehr durch den sequentiellen Anteil eines Programms limitiert.

2.7.1 Einzelne Anfrage

Der Ablauf des Retrieval-Prozesses bei NETSPEAK ist in Abbildung 2.2 dargestellt. Aus dem UML-Diagramm geht hervor, dass die Prozesse der Spell-Correction und der Query-Expansion gleichzeitig stattfinden. Dies ist möglich, da die Rechtschreibkorrektur nicht

⁵ <http://code.google.com/apis/ajaxsearch>

direkt in die Ergebnisberechnung mit eingeht, sondern nur auf der Antwortseite eingeblendet wird. Das hierbei verwendete SmartSpell, läuft als separater Dienst auf dem lokalen Server und wird in einem eigenen Thread über eine TCP/IP-Verbindung angefragt. In der Regel terminiert dieser Thread noch lange bevor die Query-Expansion beendet ist. Es gilt also Letzteres zu beschleunigen.

Für den Teil der Synonymsuche wird die Bibliothek JAWS verwendet, die sich selbst als „high-performance API“ bezeichnet. JAWS greift für eine Anfrage auf das lokale Dateisystem zu, um die WordNet Datenbank auszulesen. Ein Vergleich zu *JWNL*⁶, (Java WordNet Library) eine auf gleiche Weise funktionierende API, zeigte tatsächlich einen Geschwindigkeitsvorteil um den Faktor zwei bis drei. Beide Bibliotheken benötigen jedoch mindestens einen Zugriff auf die Festplatte, was allgemein als teure Operation gilt (vgl. Kapitel 3). JWNL bietet hingegen die Möglichkeit die gesamte WordNet Datenbank in den Speicher zu laden, um schneller darauf zugreifen zu können. Diese umfasst auf der Festplatte rund 35 Megabyte, wobei davon auszugehen ist, dass der interne Speicherverbrauch, wegen der zusätzlichen Datenstrukturen, höher liegt. Für NETSPEAK ist diese Alternative zukünftig vorgesehen.

Bei der Mustersuche im N-Gramm-Index (vgl. Abschnitt 2.4.2) finden die Lookups der Referenzlisten zur Bildung ihrer Schnittmenge statt. Da die Vorstellung der zugrundeliegenden Indeximplementierung erst in Kapitel 3 stattfindet, kann zusammenfassend gesagt werden, dass die verwendete Indextechnologie aus invertierter Liste und minimaler perfekter Hashfunktion optimale Ergebnisse in Bezug auf Speicherverbrauch und Schnelligkeit liefert. Das Kopieren der eingelesenen Postlisten des in C++ implementierten Index nach Java über das JNI ist hingegen eine zeitaufwändige Aktion, wenn dabei immer wieder neue Objekte allokiert werden. Hierbei kann Zeit gespart werden, indem einmal zugesicherter Speicher wiederverwendet wird, was mit dem JProfiler⁷ evaluiert werden kann. Dennoch bleibt eine doppelte Speicheranforderung für Postlisten auf Seiten von C++ und Java. Eine Verbesserung könnte erzielt werden, wenn die Funktion der Schnittmengenbildung von Postlisten in den Index integriert wird. Hieraus würden sich zwei Vorteile ergeben: Effizientes Operieren direkt auf den Originaldaten und weniger Speicherverbrauch, da nur die Postliste der Schnittmenge übertragen werden muss, die in der Regel sehr viel kürzer ist als die Eingabepostlisten.

⁶ <http://jwordnet.sourceforge.net>

⁷ Der JProfiler ist ein Werkzeug zur Überwachung der Ressourcennutzung von Java-Programmen zu deren Laufzeit. Es wird dazu benutzt Performanzengpässe, Speicherlecks und Probleme bei Nebenläufigkeiten aufzuspüren. <http://www.ej-technologies.com/products/jprofiler/overview.html>

Ein weiterer Ansatz ist das Zwischenspeichern von bereits berechneten Schnittmengen im Kontext der Synonymsuche (vgl. Abschnitt 2.4.1). Da aus jedem Synonym eine weitere Query generiert wird, die auch separat behandelt wird, ist es wichtig nicht mehrmals den gleichen Schnitt zu berechnen. Enthält eine Query z.B. drei Worte, von denen das letzte Wort zur Synonymsuche verwendet wird, muss der Schnitt der ersten beiden Worte nur einmalig berechnet werden. Diese Schnittmenge wird für jeden Schnitt mit der Referenzliste eines gefundenen Synonyms direkt wiederverwendet.

Beim Zusammenfassen der N-Gramme (vgl. Abschnitt 2.4.3) werden die maximal 100 ersten N-Gramme der resultierenden Referenzliste aus dem N-Gramm-Korpus eingelesen. Um bei diesem Vorgang die Eigenschaften einer Festplatte bestmöglich auszunutzen, finden diese Zugriffe asynchron statt. Im Gegensatz zum herkömmlichen synchronen I/O, bei der eine einzelne Leseoperation solange blockiert, bis die Daten komplett geladen sind, werden hierbei Zugriffe in separaten Threads ausgeführt. Bei einem Aufruf wird in der Regel ein Handler-Objekt zurückgeliefert, über welches der Status und die Ergebnisse der Operation mittels Polling abgefragt werden können. Die Unterstützung des asynchronen I/O ist dabei abhängig vom Betriebssystem bzw. vom Kernel. Linux unterstützt es ab dem Kernel 2.4 und Windows über die Win32-API ab Windows2000. Für die Verwendung in Java sind wiederum Bibliotheken notwendig, die die entsprechenden Funktionen kapseln. Für NETSPEAK unter Windows wird hierbei *AIO4J*⁸ (Asynchronous IO for Java) verwendet und unter Linux ist der Einsatz von *Asynchronous File IO in Java*⁹ aus dem Apache-MINA-Projekt¹⁰ geplant.

2.7.2 Parallele Anfragen

Die bisher besprochenen Maßnahmen dienen dazu eine einzelne Anfrage möglichst schnell zu verarbeiten. Bei einem Web-Dienst spielt weiterhin die Beantwortung gleichzeitig eintreffender Anfragen eine wesentliche Rolle. Um dieser Herausforderung bereits auf Prozessebene zu begegnen läuft der NETSPEAK-Dienst in Form eines Java-Servlets. Diese Technologie erlaubt eine bessere Performanz im Gegensatz zum herkömmlichen CGI (Common Gateway Interface), da Anfragen in einzelnen Threads, anstatt in separaten Prozessen, parallel abgehandelt werden.

Die wichtigsten Ressourcen eines Server-Systems, wie Prozessor, Hauptspeicher und Festplatte sind dagegen nur einmal vorhanden. Wenn im Zuge vieler Anfragen mehrere Threads gleichzeitig den Speicher benutzen ist dieser schnell voll und konkurrierende Lesezugriffe auf die Festplatte finden letztendlich auch seriell statt. Die Folge ist, dass das System schnell überlastet ist und keine weiteren Anfragen mehr beantworten kann.

⁸ <http://www.alphaworks.ibm.com/tech/aio4j>

⁹ <http://mina.apache.org/asynchronous-file-io-in-java.html>

¹⁰ <http://mina.apache.org/>

Die mehrfache Ausführung solcher Komponenten kann dieses Problem lösen, was allgemein als Redundanz bezeichnet wird. Hierzu gibt es u.a. zwei Ansätze: Die Vervielfachung von Festplatten innerhalb eines Rechners durch ein RAID-System und die Vernetzung von mehreren Rechnern zu einem Cluster.

RAID (Redundant Array of Independent Disks) bezeichnet einen Verbund von eigenständigen Festplatten mit redundanter Datenhaltung. Es gibt hierbei viele Varianten ein RAID aufzubauen, um entweder die Ausfallwahrscheinlichkeit zu minimieren oder den Datendurchsatz bei Lese-/Schreibzugriffen zu erhöhen. Für NETSPEAK wäre eine Kombination von gespiegelten Festplatten denkbar, die jeweils den gesamten Index und die Google-N-Gramm-Kollektion enthalten. Durch einen RAID-Controller werden dann parallele Zugriffe auf die physikalischen Festplatten verteilt, wobei deren Verbund auf Betriebssystemebene weiterhin als ein logisches Laufwerk dargestellt wird.

Beim Aufbau eines Computerclusters werden komplette Rechereinheiten miteinander vernetzt. Nach [Slo04] werden heutzutage zwei Arten von Clustern unterschieden: High-Availability-Cluster (Hochverfügbarkeitscluster) und Load-Balancing-Cluster (Lastverteilungscluster). Ersteres wird für kritische Anwendungen eingesetzt, bei denen die Ausfallsicherheit eine wichtige Rolle spielt. Hierbei kann jeder Einzelrechner den gesamten Dienst übernehmen, falls die rechnende Einheit ausfällt. Die Strategie hinter Lastverteilungscluster verfolgt dagegen einen Divide-and-Conquer-Ansatz, um die mehrfach vorhandenen Ressourcen zu nutzen. Der Serverrechner verteilt dabei eintreffende Anfragen auf einzelne Clientrechner nach einem Round-Robin-Verfahren. Durch die somit erreichte echte Parallelität, kann die Rechenleistung und die Kapazität des Hauptspeichers je nach Anzahl der Einheiten vervielfacht werden.

In der Praxis werden üblicherweise Kombinationen von RAID- und Clustersystemen eingesetzt, wie es am Beispiel der Google-Suchmaschine bei Wikipedia [Wik08b] nachzulesen ist. Google betreibt demnach weltweit eine Reihe von Clustern, die jeweils die komplette Funktionalität der Suchmaschine übernehmen können. Innerhalb eines Clusters werden Anfragen hochgradig parallel verarbeitet, wodurch sehr geringe Antwortzeiten realisiert werden können.

Auch für den NETSPEAK-Dienst ist bereits ein kleiner Linux-Cluster im Aufbau. Eine darauf laufende Middleware soll eine effiziente Lastverteilung realisieren. In Zukunft kann somit eine große Anzahl von Benutzern, mit akzeptablen Antwortzeiten, bedient werden.

3 Indizierung großer Datenmengen

Dieser Teil widmet sich der theoretischen und praktischen Realisierung eines Index. Unter einem Index versteht man eine Datenstruktur, die einen effizienten Zugriff auf eine große Menge von Daten gestattet. Innerhalb eines IR-Systems dient dieser Index dem Zugriff auf eine Wissensdatenbank, die zur Beantwortung einer Query benötigt wird. Der Prozess der Indizierung bezeichnet die Generierung dieses Index für eine bestimmte Menge von Daten. Dazu legt ein Index für jedes Datum dieser Menge einen Schlüssel mit einer dazugehörigen Position in einer Tabelle ab. Mehrere Datenelemente können dabei den gleichen Schlüssel besitzen. Eine Suche im Index findet unter Angabe eines Schlüssels statt, um die Positionen zu erfragen, an denen die zugehörigen Daten zu finden sind. Ein Index ist daher eine Funktion, die Datenschlüssel auf eine Menge von Positionen abbildet.

Die Anfragesprache eines IR-System bestimmt hierbei die Wahl einer logischen Datenstruktur zur Realisierung dieser Funktion. Die gängigsten Strukturen im Text-IR sind die invertierte Liste, die Signaturliste, Suffixbäume und Suffix-Arrays [BYRN99]. Für den NETSPEAK-Dienst wird die invertierte Liste zur Indizierung der Google-N-Gramm-Kollektion angewandt, da sich diese für den gegebenen Anwendungsfall am besten eignet.

Eine sehr wichtige Rolle spielt die Implementierung der gewählten Datenstruktur. Es ist zu beachten, dass die Menge der im Index gespeicherten Positionen, bei wortgenauer Indizierung, meist die gleiche Größenordnung besitzt, wie die Menge der indizierten Daten. Bei großen Datenmengen kann die Organisation der Indexstruktur somit nicht vollständig im Hauptspeicher stattfinden, sondern sie muss auf einen externen Massenspeicher erweitert werden. Daraus folgt, dass sich zur Laufzeit eines IR-Systems ein Teil der Indexstruktur im Hauptspeicher befindet, während die Indexdaten auf dem Medium liegen. Eine invertierte Liste kann auf diese Weise als verteilte Hashtabelle oder B-Baum implementiert werden.

Die Wahl der konkreten Datenstruktur hängt davon ab, ob es sich eher um einen statischen oder dynamischen Index handelt. Statische Indexe sind dadurch gekennzeichnet, dass zur Laufzeit keine oder nur sehr vereinzelte Einfüge- und Löschoptionen von Datenelementen stattfinden. Hierfür ist die Hashtabelle einer baumbasierten Struktur vorzuziehen. Das Gegenteil ist bei dynamischer Indizierung der Fall. Der NETSPEAK-Dienst verwendet eine Hashtabelle zur statischen Indizierung der N-Gramm-Kollektion.

Hashtabellen haben den Vorteil, dass sie im Allgemeinen die beste Zugriffsperformanz auf die indizierten Daten bieten. Eine sehr wichtige Rolle spielt in diesem Zusammenhang

die Anzahl der externen Zugriffe, die direkte Auswirkung auf die Antwortzeit eines IR-Systems hat. Eine Grundvoraussetzung ist hierbei ein wahlfreier Zugriff auf das Medium, der, im Gegensatz zu Bandlaufwerken, bei Magnetfestplatten möglich ist. Dennoch ist festzustellen, dass heutige Festplatten mit 10.000 Umdrehungen pro Minute im Vergleich zum Hauptspeicher eine 120.000-mal¹ langsamere Zugriffszeit aufweisen. Eine Herausforderung bei der Konstruktion eines Index ist es deshalb, die Festplattenzugriffe pro Anfrage zu minimieren. Ein einzelner sogenannter Index-Lookup ist bei Hashtabellen in konstanter Zeit möglich.

3.1 Datenstruktur der invertierten Liste

Die invertierte Liste ist eine sehr häufig angewandte Struktur zur Indizierung von Daten. Im textbasierten Information-Retrieval handelt es sich bei diesen Daten meist um Textdokumente, in denen später nach Worten oder Phrasen gesucht werden soll. In diesem Zusammenhang entspricht ein Schlüssel im Index einem Wort oder Term in diesen Textdokumenten. Da auf ähnliche Weise auch mediale Inhalte wie Bilder, Audio und Video indiziert werden können, wird im Folgenden weiterhin der abstrakte Begriff *Daten* verwendet. Bei einer Indizierung geht es auch vielmehr um die Art der Adressierung, so dass von den eigentlichen Inhalten abstrahiert werden kann.

Der logische Aufbau einer invertierten Liste ist in Abbildung 3.1 in Form einer zweiseitigen Tabelle dargestellt. Jede Zeile enthält einen Term mit einer dazugehörigen Liste von Referenzen. Die Menge aller Terme wird als Vokabular oder Lexikon eines Index, bzw. auch als Indexterme bezeichnet. Die einem Term zugeordnete Liste wird Postliste genannt. Es handelt sich also um eine Abbildung die jedem Term aus einem Vokabular eine Menge von Referenzen zuordnet.

Vokabular	Postlisten
Term ₁	Referenz ₁₁ , Referenz ₁₂ , Referenz ₁₃
Term ₂	Referenz ₂₁ , Referenz ₂₂
Term ₃	Referenz ₃₁ , Referenz ₃₂
...	...
Term _n	Referenz _{n1} , Referenz _{n2} , Referenz _{n3}

Tabelle 3.1: Struktur einer invertierten Liste.

¹ Angenommen werden eine Festplatte mit 10.000 UPM und Hauptspeicher mit $t_{mem} = 25ns$ Zugriffszeit. Die Zeit für einen Festplattenzugriff entspricht im Durchschnitt einer halben Festplattenumdrehung $t_{hdd} = 0,5 \times 60/10.000 = 0,003s$. Der Faktor beträgt demzufolge $d = t_{hdd}/t_{mem} = 120.000$.

Zur Verdeutlichung sei an den Index eines Buches erinnert. Hier findet sich eine Auflistung von relevanten Suchbegriffen, denen jeweils Seitenzahlen zugeordnet sind. Ein anderes Beispiel ist eine Internet-Suchmaschine, bei der vereinfacht ausgedrückt, jedes Suchwort auf eine Liste von Internet-Adressen verweist. Die Granularität der Adressierung kann von Fall zu Fall sehr unterschiedlich sein. Bei dem Buchindex und der Suchmaschine werden ganze Dokumente referenziert, während bei einer wortgenauer Indizierung die Position eines Wortes in einem bestimmten Textdokument gespeichert wird. Ein Eintrag der Postliste würde dann einem Tupel aus Dokumentreferenz und Wortposition entsprechen. Neben der Speicherung dieser Art von Referenzen ist es auch möglich Metainformationen, wie etwa die Häufigkeit eines Terms, in einer Postliste abzulegen.

Das Vokabular eines Index erschließt sich nicht immer unmittelbar aus den zu indizierenden Daten, sondern hängt von der Granularität der Indizierung ab. Bei einer wortweisen Indizierung entspricht das Vokabular allen im Text vorkommenden Worten, während bei dokumentweiser Indizierung repräsentative Schlüsselworte zunächst bestimmt werden müssen. Zur automatischen Extraktion solcher Begriffe bietet das Information-Retrieval entsprechende Verfahren an. Bei medialen Inhalten ist es dagegen notwendig die Bild-, Musik- oder Videodaten vor einer Indizierung manuell mit Schlüsselworten zu versehen.

Da es sich bei dem Vokabular um diejenigen Terme handelt, die später zur Suche im Index verwendet werden, ist auch die Groß- und Kleinschreibung von Bedeutung. Angenommen es gäbe einen Index der die beiden Terme `netspeak` und `NETSPEAK` als Schlüssel enthält. Eine Anfrage an diesen Index mit dem Term `NetSpeak` würde zu keinem Ergebnis führen, da dieser nicht Teil des Vokabulars ist. Werden dagegen alle Terme mit einer einheitlichen Konvertierung indiziert, führen Anfragen nach `netspeak`, `NETSPEAK` oder `NetSpeak` auf dieselbe Postliste. Bei der vorliegenden Indeximplementierung kann mit oder ohne Unterscheidung der Groß- und Kleinschreibung indiziert werden. Bei der Indizierung der N-Gramm-Kollektion für den `NETSPEAK`-Dienst findet eine einheitliche Konvertierung statt.

3.1.1 Entfernung von Stoppworten

Als Stoppworte werden Worte bezeichnet, die sehr häufig in allen Dokumenten auftreten, aber in Bezug auf den Inhalt eines einzelnen Dokuments nur einen geringen Informationswert besitzen. Das heißt, dass diese Worte die Semantik eines Dokuments nicht adäquat repräsentieren und somit keine klare Abgrenzung zu anderen Dokumenten gewährleisten. Sie eignen sich daher nicht als Indexterme. Zu den Stoppworten zählen Artikel, Präpositionen, Konjunktionen, einige Verben, Adverbien und Adjektive sowie speziell geschlossene Wortklassen, wie Zahlworte, Monate, Währungen und Namen.

Der absolute Anteil von Stoppworten im Vokabular einer Sprache umfasst zwar nur 200 bis 1000 Worte, doch ihr relativer Anteil in einer Textkollektion beträgt etwa 25 Prozent

Rang	Term	Häufigkeit	Rang	Term	Häufigkeit
1	the	69971	1	ein	710719
2	of	36411	2	und	708531
3	and	28852	3	in	613869
4	to	26852	4	sein	534056
5	a	23237	5	werden	400264
6	in	21347	6	haben	340313
7	this	10099	7	von	333335
8	is	10019	8	zu	290911
9	what	9816	9	mit	286015
10	he	9543	10	im	278227

Tabelle 3.2: Häufigkeiten von Stoppworten in zwei Textkollektion.

[WMB94]. Durch eine Indizierung ohne Stoppworte kann demzufolge die Indexstruktur um ein Viertel verkleinert werden. Tabelle 3.2 zeigt die zehn häufigsten Terme aus einer Stichprobe von 1.000.000 Worten aus dem englischen Brown-Korpus (links) und aus einer Stichprobe von 3.000.000 Worten aus dem Korpus der Stuttgarter Zeitung (rechts) [Ste07].

In Hinblick auf das Suchergebnis hat die Entfernung von Stoppworten allerdings auch Nachteile. Einige Stoppworte haben Homonyme, die durchaus als Indexterme geeignet wären. Das englische Verb `can` (dt.: können) ist ein Stoppwort, besitzt jedoch eine zweite Bedeutung als Substantiv (dt.: Becher, Kanne). Bei einer Indizierung ohne Stoppworte, liefert daher eine Suche nach `can` in beiden Fällen keine Dokumente zurück.

Ein weiteres Problem tritt bei der Suche nach Phrasen auf, die hauptsächlich aus Stoppworten bestehen. Die Boolean-Query „*to be or not to be*“ sollte diejenigen Dokumente liefern in denen alle Terme gemeinsam auftreten und darin mittels Mustersuche die exakte Phrase lokalisieren. Da es sich bei allen Worten um Stoppworte handelt, die nicht indiziert wurden, bliebe diese Suche allerdings erfolglos. Die meisten Internet-Suchmaschinen sowie der NETSPEAK-Dienst verzichten daher auf eine Stoppwortentfernung.

3.1.2 Reduktion auf den Wortstamm

Dieser Prozess dient in erster Linie dazu, die Anzahl zurückgelieferter Dokumente, zu erhöhen. Da viele Worte in einem Textdokument in flektierter Form vorliegen, ist es sinnvoll, nicht jede Variation als Indexterm aufzunehmen. Angenommen ein Benutzer sucht nach dem Wort `index`, so würde er nicht automatisch auch die Dokumente für `indexes`, `indexed` und `indexing` geliefert bekommen. Um für Worte mit gleichem Wortstamm eine gemeinsame Postliste zu generieren, werden flektierte Worte während einer Indizierung auf ihren Wortstamm reduziert und nur dieser als Indexterm aufgenommen.

<i>sses</i>	→	<i>ss</i>
<i>ies</i>	→	<i>i</i>
<i>ing</i>	→	ε
<i>ed</i>	→	ε
<i>ly</i>	→	ε

Abbildung 3.1: Ersetzungsregeln des Porter-Stemmer-Algorithmus.

Die Rückführung einer flektierten Form auf ihren Wortstamm erfordert die Entfernung von Vorsilben und Nachsilben, die bei der Deklination von Substantiven, der Konjugation von Verben und der Komparation von Adjektiven auftreten. Es existieren verschiedene Verfahren um diese sprachspezifischen Affixe zu erkennen und zu entfernen. Eine Möglichkeit bietet eine statische Hashtabelle in der vorab alle möglichen Flexionen aufgenommen und auf ihren Wortstamm abgebildet werden. Methoden der strukturellen Linguistik arbeiten mit der Bestimmung von Morphemgrenzen. Die zeichenbasierte N-Gramm-Analyse versucht durch Windowing typische Bi- und Trigramme zu identifizieren. Als gebräuchlichstes Prinzip gilt allerdings die regelbasierte Ersetzung bzw. Entfernung von Affixen. Hierbei wird mittels eines Zustandsautomaten eine Menge von Ersetzungsregeln auf jedes Wort angewandt. Der bekannteste Algorithmus dieser Art ist der Porter-Stemmer, da er sehr gute Ergebnisse liefert. Abbildung 3.1 zeigt einige einfache Regeln zur Ersetzung von Endungen, wobei ε das leere Wort bezeichnet [BYRN99].

Eine Schwäche dieses Algorithmus ist jedoch sein Potential zur Übergeneralisierung. Außerdem ist eine Wortstammreduktion generell nicht angebracht, wenn nach exakten Phrasen gesucht werden soll, so dass viele Internet-Suchmaschinen sowie der NETSPEAK-Dienst darauf verzichten.

3.2 Implementierung eines invertierten Index

Für die Indizierung der Google-N-Gramm-Kollektion wurde eine eigene Implementierung in der Programmiersprache C++ entwickelt. Dieser Index arbeitet nach dem Prinzip der invertierten Liste und liefert für unsere Anforderungen eine sehr gute Retrieval-Performanz. Durch sein erweiterbares Design und einem generischen Eingabeformat ist er außerdem als Bibliothek für beliebige Indizierungsaufgaben geeignet. Mit dem Java-Native-Interface (JNI) wurde weiterhin eine Schnittstelle für Java-Applikationen zur Verfügung gestellt. Im Anhang dieser Arbeit befindet sich der Quellcode sowie die Dokumentation zum Index.

3.2.1 Eingabeformat: Pseudo-invertierte Liste

Das Eingabeformat dieser Implementierung ist eine invertierte Liste in Textrepräsentation, aufgeteilt auf eine oder mehrere Textdateien. Exakter formuliert, handelt es sich um eine pseudo-invertierte Liste, da es erlaubt ist, dass Indexterme mehrfach auftreten können. Das bietet den Vorteil, dass die zu indizierenden Daten sequentiell geparkt, und keine Informationen über bereits vorhandene Schlüssel im Hauptspeicher gehalten werden müssen. Letzteres wäre bei begrenztem Hauptspeicher unpraktikabel. Da also zu keiner Zeit die gesamte Postliste für einen Indexterm vorliegt, wird diese abschnittsweise definiert.

Die Struktur einer Textdatei der pseudo-invertierten Liste ist in Tabelle 3.3 dargestellt. Jede Zeile beginnt mit einem Indexterm gefolgt von einer Liste mit Referenzen. Der Indexterm $Term_2$ tritt hierbei mehrfach auf. Das Format der Referenz kann zur Laufzeit gewechselt werden, ohne dass der Index selbst neu kompiliert werden muss.

Die Erzeugung dieser Textdateien, durch Parsen der zu indizierenden Daten, ist ein Vorprozess und nicht Teil der Indeximplementierung. Sie muss vom Benutzer des Index selbst durchgeführt werden, da jede Kollektion von Daten individuelle Eigenschaften mitbringt, auf die nur mit großem Aufwand automatisch reagiert werden kann. Aufgabe des Index ist es, Postlisten mit gleichen Indextermen zu vereinigen und die entstandene echte invertierte Liste in eine Datenstruktur zu überführen, auf die effizient zugegriffen werden kann. Im Folgenden werden die zu indizierenden Daten als Nutzdaten bezeichnet.

Neben Nutzdaten kann dieser Index auch Metadaten indizieren. Die Textdateien müssen dafür bereits als echte invertierte Liste vorliegen, d.h. es dürfen keine Indexterme doppelt auftreten. Zudem werden nur diejenigen Indexterme betrachtet, die auch Teil des Vokabulars der Nutzdaten sind, es wird also auch zwischen Klein- und Großschreibung unterschieden. Die Einträge einer Postliste werden stets als String-String-Tupel interpretiert und sind nicht variabel. Der Zweck dieser Definition liegt in der Möglichkeit einem Term eine Liste von Schlüssel-Wert-Paaren zuzuordnen, die beim Lookup eines Terms als Properties-Objekt zurückgeliefert werden.

Vokabular	Postlisten
Term ₁	Referenz ₁₁ , Referenz ₁₂ , Referenz ₁₃
Term₂	Referenz₂₁, Referenz₂₂
Term ₃	Referenz ₃₁ , Referenz ₃₂
...	...
Term₂	Referenz₂₃, Referenz₂₄, Referenz₂₅
...	...
Term _n	Referenz _{n1} , Referenz _{n2} , Referenz _{n3}

Tabelle 3.3: Struktur einer pseudo-invertierten Liste.

```

options to deal with this 1493
options to deal with those 50
options to decide how big 59
options to decide what is 101

```

Abbildung 3.2: Ausschnitt einer Textdatei aus dem 5-Gramm-Korpus.

Beispiel: Parsing der Google-N-Gramm-Kollektion

Zur Indizierung der Google-N-Gramme stellt der Index einen Parser bereit, der die Dateien der Google-N-Gramm-Kollektion in das Format der Nutz- und Metadaten überführt. Die Textdateien dieser Kollektion sind zeilenweise aufgebaut. Jede Zeile beginnt mit einem N-Gramm - je nach Korpus also ein bis fünf Terme - gefolgt von einem Häufigkeitswert, der die absolute Anzahl des N-Gramms in den von Google indizierten Web-Seiten angibt. Alle Einträge sind durch Leerzeichen getrennt. Abbildung 3.2 zeigt einen Ausschnitt einer solchen Datei aus dem 5-Gramm-Korpus.

NETSPEAK verwendet diesen 5-Gramm-Korpus als Nutzdaten und den 1-Gramm-Korpus als Metadaten. Die Menge der 5-Gramme dient dazu, Wildcard-Queries an den NETSPEAK-Dienst bis zu einer Länge von fünf Termen im Ergebnis zu expandieren. Die 1-Gramme liefern den absoluten Häufigkeitswert zu einem Term, der für das Ranking benötigt wird. Eine Indizierung der 5-Gramme bedeutet, bei einer Suche nach einem Term, eine Liste mit Referenzen auf diejenigen 5-Gramme zu erhalten, in denen dieser Term vorkommt. Dazu verweist eine Referenz in der Postliste genau auf ein 5-Gramm im 5-Gramm-Korpus.

Das Format dieser Referenz ist in Abbildung 3.3 in EBNF-Notation dargestellt. Es handelt sich um ein 5-Tupel bestehend aus Datei-Id, Offset, Bytelänge und Häufigkeitswert eines 5-Gramms sowie einer Positionsangabe des betrachteten Terms innerhalb dieses 5-Gramms. Dieses 5-Tupel wird im Folgenden abkürzend als GNGramPointer bezeichnet. (Hinweis: Bei alle Elementen des GNGramPointer handelt es sich um positive Zahlen. Beim Parsen werden diese jedoch als vorzeichenbehaftete Integer interpretiert. Dieses Vorgehen geschieht aus Kompatibilitätsgründen zur Programmiersprache Java, bei der keine vorzeichenlosen Datentypen existieren. Es steht somit nur der halbe Wertebereich eines Integers zur Verfügung. Ein Minuszeichen wird durch die Grammatik ausgeschlossen.)

```

GNGRAMPOINTER = FID OFF LEN CNT POS ;
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
NUMBER = DIGIT { DIGIT } ;
FID = NUMBER ;           // 8 Bit signed Integer [0, 127]
OFF = NUMBER ;          // 32 Bit signed Integer [0, 2.147.483.647]
LEN = NUMBER ;          // 8 Bit signed Integer [0, 127]
CNT = NUMBER ;          // 32 Bit signed Integer [0, 2.147.483.647]
POS = NUMBER ;          // 8 Bit signed Integer [0, 127]

```

Abbildung 3.3: Format eines GNGramPointer in EBNF-Notation.

```

options 93 284521063 30 1493 0 93 284521094 29 50 0 \
 93 284521124 28 59 0 93 284521153 29 101 0
to 93 284521063 30 1493 1 93 284521094 29 50 1 \
 93 284521124 28 59 1 93 284521153 29 101 1
deal 93 284521063 30 1493 2 93 284521094 29 50 2
with 93 284521063 30 1493 3 93 284521094 29 50 3
this 93 284521063 30 1493 4
those 93 284521094 29 50 4
decide 93 284521124 28 59 2 93 284521153 29 101 2
how 93 284521124 28 59 3
big 93 284521124 28 59 4
what 93 284521153 29 101 3
is 93 284521153 29 101 4

```

Abbildung 3.4: Ausschnitt einer (pseudo-) invertierten Liste.

Ein Parser liest zeilenweise jedes 5-Gramm der 118 Dateien des 5-Gramm-Korpus (vgl. 1.1) und erzeugt für jeden darin enthaltenen Term einen GNGramPointer, d.h. bis zu fünf Stück pro Zeile. Für Terme, die nicht indiziert werden sollen, wie etwa Stoppworte, wird kein GNGramPointer generiert. Ein ganzes 5-Gramm wird hingegen verworfen, wenn es mindestens ein Sonderzeichen enthält. Ein Sonderzeichen ist hierbei definiert als ein nicht-alphanumerisches Zeichen. Ausnahmen bilden die beiden Interpunktionen Komma und Apostroph, die mit indiziert werden. Diese Filterung trägt zur Reduktion der Länge der Postlisten bei, da viele 5-Gramme schlichtweg unbrauchbar sind².

Aus Effizienzgründen werden die auf diese Weise generierten Paare aus Term und GNGramPointer nicht sofort als neue Zeile in die Textdatei(en) der pseudo-invertierte Liste geschrieben, sondern zunächst dateiweise im Hauptspeicher zu kleinen Postlisten organisiert, die dann periodisch auf einmal geschrieben werden. Dieses Vorgehen erfordert weniger Festplattenzugriffe und erzeugt kleinere Dateien, da nicht jede Zeile nur aus einem Term-GNGramPointer-Paar besteht.

Abbildung 3.4 zeigt den entsprechenden Ausschnitt dieser invertierten Liste für das Beispiel 3.2. Der erste GNGramPointer der ersten Zeile verweist demzufolge auf ein 5-Gramm, das sich in der 93. Datei des 5-Gramm-Korpus an einem Offset von 284.521.063 Bytes befindet. Dieses 5-Gramm besteht aus 30 Zeichen und besitzt eine absolute Häufigkeit von 1493. Weiterhin steht der Term `options` an Position 0 innerhalb dieses 5-Gramms.

Die Konvertierung des 1-Gramm-Korpus in das Format der Metadaten erfolgt auf ähnliche Weise. Prinzipiell wird jede Zeile, bestehend aus Term und Häufigkeitswert, direkt in eine Zeile aus Term und Schlüssel-Wert-Paar umgewandelt. Abbildung 3.5 demonstriert das Vorgehen für eine Indizierung (a) mit und (b) ohne Berücksichtigung der Groß- und Kleinschreibung. Bei Letzterem werden die Häufigkeitswerte verschiedener Schreibweisen

² Betrifft 5-Gramme mit Datum-/Zeitangaben, URLs, HTML-/XML-Entities usw., die in einer Vielzahl von Kombinationen auftreten (z.B. § 4732.00 – § 5754.00).

(a)	indexing	2203365	→	indexing	count	2203365	
	Indexing	1202159	→	Indexing	count	1202159	
(b)	indexing	2203365	}		indexing	count	3405524
	Indexing	1202159					

Abbildung 3.5: Beispielkonvertierung zur Erzeugung von Metadaten.

eines Terms erst addiert und dann in die Metadatei geschrieben. Da die 1-Gramm-Datei nur 177 Megabyte groß ist, kann dieser Prozess im Hauptspeicher stattfinden.

Am Ende der beschriebenen Konvertierungen stehen 118 Dateien an Nutzdaten in Form einer pseudo-invertierten Liste und eine Metadatei für den Indizierungsprozess bereit. Die Nutzdaten haben eine Gesamtgröße von rund 30 Gigabyte und beinhalten etwa 2,7 Milliarden GNGramPointer.

3.2.2 Indexkomponenten und ihre Funktionen

Der invertierte Index ist je nach Repräsentation der Daten logisch in drei Schichten unterteilt, die in Abbildung 3.6 dargestellt sind. Auf der untersten Schicht, dem Interpreted-Index-Data-Layer, wird durch eine Konfigurationsdatei eine *ComponentFactory* initialisiert, die alle konkreten Komponenten des Index erzeugt. Über eine Instanz vom Typ *DataSource* findet die Bereitstellung der Nutz- und Metadaten statt. Die zu indizierenden Datenelemente müssen hier exakt interpretiert werden, um etwa das Parsing von Textdateien einer invertierten Liste zu ermöglichen.

Auf der mittleren Ebene, dem Uninterpreted-User-Data-Layer, werden diese Dateneinheiten nur noch neutral als *ByteArrays* betrachtet. Diese werden hier zu *GenericPostlists* zusammengeführt, die wiederum in einem *DataStorage* organisiert werden. Das *DataStorage* kapselt dabei zusammen mit einer *HashFunction* die eigentliche Indexdatenstruktur. Der *GenericIndex* bietet hier bereits eine Schnittstelle zum Lookup des Index an, um *GenericPostlists* mit *ByteArrays* auszulesen. Diese neutrale Repräsentation dient der generischen Anbindung des Index an andere Softwarekomponenten. An dieser Stelle findet deshalb die Übertragung von Postlisten zu Java über das Java-Native-Interface für den NETSPEAK-Dienst statt. Details dazu werden in Abschnitt 3.2.7 genauer erläutert.

Die oberste Schicht, der Interpreted-User-Data-Layer, repräsentiert die Benutzerschnittstelle, auf der die Dateneinheiten in den Postlisten wieder korrekt interpretiert werden. Hierfür wird ein *InvertedIndex*-Template mit einem konkreten *PostlistEntry* instanziiert. Der *InvertedIndex* dient dann als Wrapper eines *GenericIndex* und liefert *TypedPostlists* bei einem Lookup zurück.

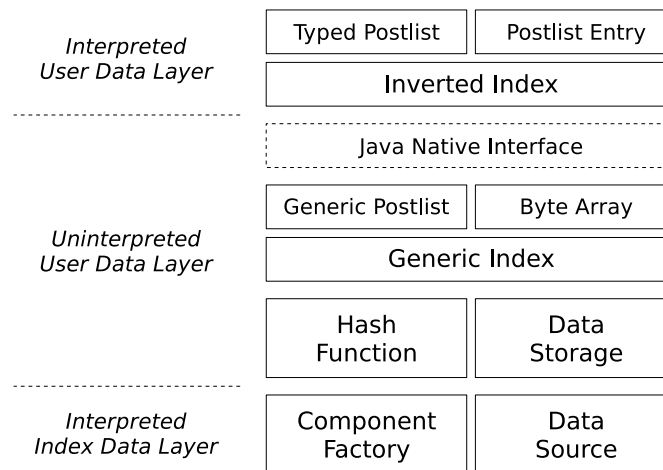


Abbildung 3.6: Logischer Aufbau der Indexkomponenten.

Bei den Hauptkomponenten *DataSource*, *DataStorage*, *HashFunction* und *PostlistEntry* handelt es sich um abstrakte Klassen ohne Implementierung. Sie definieren lediglich eine bestimmte Funktionalität innerhalb des Index, die durch Ableitung unterschiedliche Realisierungen erlaubt. Im Folgenden werden die wichtigsten Klassen des Index kurz vorgestellt, deren Beziehungen zueinander im UML-Diagramm 3.8 dargestellt sind. Eine Beschreibung der genauen Schnittstellen befindet sich in der Dokumentation im Anhang B.2.

ComponentFactory Dieses Objekt erzeugt alle konkreten Ableitungen der Basisklassen *PostlistEntry*, *DataSource*, *DataStorage* und *HashFunction*. Dazu erhält es bei seiner Instanziierung die Konfigurationsdatei des Index. Die Factory kennt alle zu erwartenden Schlüssel dieser Datei und überprüft diese auf Vollständigkeit. Ist dieser Test erfolgreich kann über entsprechende Methoden die jeweilige Komponente erstellt werden. Jede Komponente wird noch innerhalb dieser Factory mit ihren spezifischen Parametern initialisiert.

PostlistEntry Diese Basisklasse repräsentiert das Referenzobjekt einer Postliste im Interpreted-Index-Data-Layer und im Interpreted-User-Data-Layer. Sie wird in Situationen verwendet, in denen einzelne Elemente eines Referenzobjekts modifiziert werden müssen oder eine objektabhängige Funktionalität erforderlich ist. So benötigt der *InvertedFileParser* (*DataSource*) eine Instanz dieses Objektes, um die Eingabedatei der pseudo-invertierten Liste korrekt interpretieren zu können und die Sortierungsmethode der *GenericPostlist*, zur Definition einer Ordnungsrelation über den Elementen einer Postliste. Abgeleitete Klassen müssen daher die Überladung des Input- und Output-Operators sowie der relationalen Operatoren implementieren. Weiterhin werden spezielle Setter und Getter für einzelne Elemente des Referenzobjekts zur Verfügung gestellt.

PostlistEntry stellt zudem eine Schnittstelle zur *ByteArray*-Serialisierung bzw. zur Deserialisierung eines solchen bereit. Ersteres findet im Anschluss an das Parsing, beim Übergang

vom Interpreted-Index-Data-Layer zum Uninterpreted-User-Data-Layer, statt. Letzteres wird bei einem Index-Lookup auf Benutzerebene angewandt, um *ByteArrays* als konkrete *PostlistEntry* zur Verfügung zu stellen.

Bei vorliegender Implementierung sind bereits eine Reihe von abgeleiteten Klassen vorhanden, die für die meisten Indizierungsaufgaben ausreichend sein dürften. Dazu gehören Klassen-Templates für 1-, 2-, 3-, 4- und 5-Tupel aus primitiven Typen (*short*, *int*, *long*, *float*, *double*) sowie für String-Primitive- und String-String-Tupel. Weitere Klassen können bei Bedarf mit geringem Aufwand implementiert werden.

DataSource Diese Komponente ist für das Einlesen oder Generieren der Eingabedaten zuständig. Sie liefert Objekte vom Typ *DataItem*. Ein *DataItem* ist ein Tripel bestehend aus einem Hashwert, einer Prüfsumme und einem *ByteArray*. Hashwert und Prüfsumme werden aus einem Indexterm mittels zweier Hashfunktionen berechnet und das *ByteArray* ist die Serialisierung eines *PostlistEntry*. Die *ByteArrays* aller *DataItems* mit gleichem Hashwert werden beim Indizierungsprozess derselben Postliste hinzugefügt.

Der Hashwert dient später bei der Suche im Index dazu, die richtige Postliste zu einem gegebenen Indexterm einzulesen. Die Prüfsumme wird hingegen direkt in der Postliste gespeichert und dient der Erkennung von Hashkollisionen.

Eine konkrete Realisierung einer *DataSource* ist der *InvertedFileParser*. Diese Klasse liest Textdateien im Format der pseudo-invertierten Liste. Abbildung 3.7 demonstriert schematisch das Vorgehen, bei der aus einer Zeile der Eingabedatei *DataItems* generiert werden. Dafür wird zunächst aus dem Indexterm *options* ein Hashwert und eine Prüfsumme berechnet. Der Rest der Zeile wird mit einer Instanz eines *GNGramPointer* (*PostlistEntry*) geparkt und zu *ByteArrays* serialisiert. Jedes erzeugte *ByteArray* wird, zusammen mit dem berechneten Hashwert und der Prüfsumme, in ein *DataItem* kopiert. Auf diese Weise werden für jede Zeile mit *k* Referenzen auch *k* *DataItems* mit gleichem Hashwert und Prüfsumme generiert. Bei der pseudo-invertierten Liste für die 5-Gramm-Indizierung entstehen somit rund 2,7 Milliarden *DataItems*.

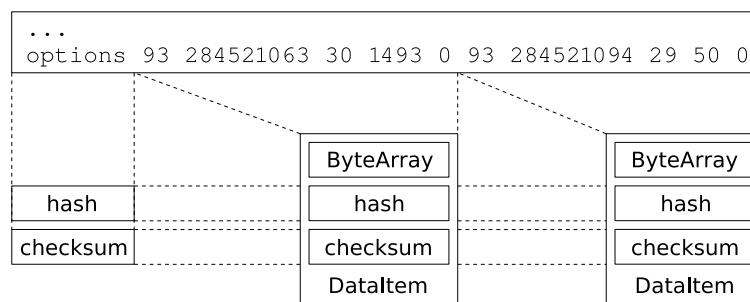


Abbildung 3.7: Erzeugung von *DataItems* aus der invertierten Liste.

HashFunction Die Hashfunktion dient dazu, jedem Indexterm einen 32 Bit großen Zahlenwert zuzuordnen. Dieser Zahlenwert wird bei der Suche im Index zur Bereitstellung der zugehörigen Postliste benötigt. Die Hashfunktion muss dabei gewährleisten, dass keine zwei Indexterme aus dem Vokabular den gleichen Hashwert erhalten, ansonsten kommt es zu einer Hashkollision. Eine Hashkollision tritt mit hoher Wahrscheinlichkeit dann ein, wenn im Index nach einem Term gesucht wird, der nicht Teil des Vokabulars ist. Auf die theoretischen Grundlagen von Hashfunktionen sowie der hier speziell eingesetzten minimalen perfekten Hashfunktion (*MPHashFunction*) wird in Abschnitt 3.2.5 im Detail eingegangen.

Neben dem Hashwert berechnet diese Komponente auch eine 16 Bit große Prüfsumme für einen Indexterm. Die Prüfsumme ist ein weiterer Hashwert, der mit einer zweiten Hashfunktion berechnet wird. Sie wird dazu verwendet um Hashkollisionen festzustellen. Die Berechnung dieser Prüfsumme findet direkt in der Basisklasse *HashFunction* statt und kann durch Ableitung nicht überschrieben werden. Zur Anwendung kommt hierbei folgende Hashfunktion aus dem Karp-Rabin-Algorithmus für einen Term x (vgl. [Sed93]):

$$h(x) = (x[0] \cdot 31^{(|x| - 1)} + x[1] \cdot 31^{(|x| - 2)} + \dots + x[|x| - 1] \cdot 31^0) \bmod 65521 \quad (3.1)$$

DataStorage Diese Komponente konstruiert und verwaltet alle Postlisten des Index. Ihr wird deshalb eine *DataSource* zugewiesen, aus der sie eine Menge von *DataItems* ausliest. Aus den *ByteArrays* aller *DataItems* mit jeweils gleichem Hashwert werden *GenericPostlists* generiert. Für *DataStorage* gibt es zwei Implementierungen: *InternalDataStorage* und *ExternalDataStorage*, die sich hinsichtlich ihrer Strategie zur Erzeugung und Vorhaltung von Postlisten unterscheiden. *InternalDataStorage* sortiert alle *DataItems* im Hauptspeicher in die jeweilige Postliste ein und hält diese auch permanent dort vor. *ExternalDataStorage* verwendet ein externes Sortierverfahren, um *DataItems* zu *GenericPostlists* zusammenzuführen. Generierte Postlisten verbleiben auch nicht im Speicher, sondern werden sofort auf Festplatte serialisiert. Jeder Index-Lookup erfordert dann einen externen Zugriff.

Welche der beiden vorgestellten Klassen konkret eingesetzt werden sollte, muss anhand der Datenmenge in Abhängigkeit des verfügbaren Hauptspeichers abgeschätzt werden. Die konkrete Klasse kann in der Konfigurationsdatei zum Index angegeben werden. Eine weitere Ableitung ist die Klasse *CachedDataStorage*, die mittels einer LFU-Strategie (Least-Frequently-Used), nur die am häufigsten angefragten Postlisten im internen Speicher behält. Sie steht allerdings bei vorliegender Implementierung noch nicht zur Verfügung.

GenericPostlist Diese konkrete Klasse repräsentiert eine Postliste im Uninterpreted-User-Data-Layer. Ihre Elemente sind vom Typ *ByteArray*. *GenericPostlists* werden in einer *DataStorage* konstruiert und verwaltet. Instanzen dieser Klasse können im internen

Speicher nur maximal 20 Megabyte (Pagegröße) einnehmen. Wächst eine Postliste bei ihrer Konstruktion über diesen Wert hinaus, erzeugt sie automatisch eine Auslagerungsdatei, in der sie ihren aktuellen Inhalt speichert. Zur permanenten Speicherung können Instanzen auch komplett auf eine Festplatte serialisiert werden. Bei einer Deserialisierung gilt dann ebenso die o.g. Limitierung. Während des Auslesens einer Postliste erfolgt jedes Mal bei Erreichen des Endes einer Page ein Swapping. Dieses Vorgehen ermöglicht die Erzeugung sehr langer Postlisten, wie sie bei Stoppworten auftreten können. Ihre Größe wird dabei nur durch die maximale Dateigröße des Dateisystems begrenzt. Postlisten weniger häufig vorkommender Worte können dagegen in der Regel komplett in den Hauptspeicher geladen werden.

Manchmal ist es notwendig die Elemente einer Postliste zu sortieren. Die *GenericPostlist* stellt dazu eine Sortierungsmethode bereit, bei deren Aufruf eine Instanz vom Typ *PostlistEntry* übergeben werden muss. Diese Instanz dient dann als Vergleichsobjekt, ein sogenannter Comparator, indem sie die *ByteArrays* wrapt und eine Überladung der relationalen Operatoren zur Verfügung stellt. In Abhängigkeit der Größe einer Postliste werden hierbei zwei verschiedene Strategien verfolgt. Eine Postliste wird komplett in den Hauptspeicher geladen und dort mittels Intro-Sort [Inc06] sortiert, wenn sie inklusive ihrer Auslagerungsdatei nicht größer als 300 Megabyte ist. Andernfalls findet ein externes Merge-Sort [Sed93] statt. Bei der 5-Gramm-Indizierung für NETSPEAK werden auf diese Weise alle *GNGramPointer* anhand ihres Häufigkeitswerts absteigend angeordnet.

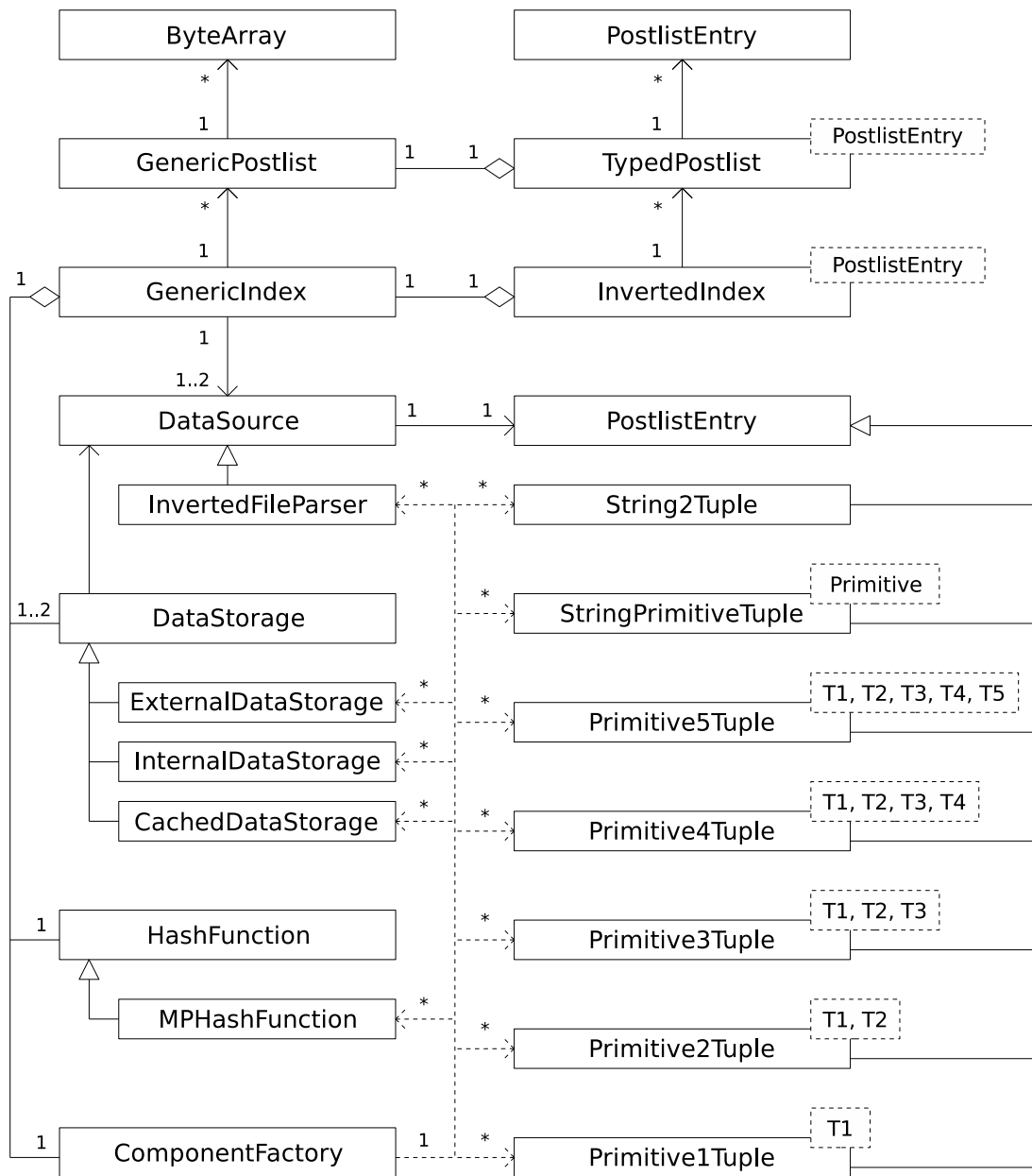


Abbildung 3.8: UML-Klassendiagramm der Hauptkomponenten.

3.2.3 Prozess der Indexkonstruktion

Dieser Abschnitt gibt einen Überblick über den gesamten Indizierungsprozess. Als Vorbereitung darauf hat der Nutzer folgende Eingabedaten in einer vordefinierten Verzeichnisstruktur bereitzustellen: Die Nutzdaten und optionalen Metadaten in einem nach Abschnitt 3.2.1 definierten Format sowie eine Konfigurationsdatei, die im Wesentlichen die konkreten Klassennamen der Indexkomponenten *PostlistEntry*, *DataSource* und *DataStorage* festlegt. Es sei noch einmal darauf hingewiesen, dass die Erzeugung dieser Eingabedateien durch einen beliebigen Vorprozess zu realisieren und nicht Teil der Indeximplementierung ist. Für den Fall der in Abschnitt 3.2.1 beschriebenen 5-Gramm-Indizierung stellt der Index entsprechende Funktionen zur Verfügung. Der Indizierungsprozess kann anschließend unter Angabe der Konfigurationsdatei aus C++ oder Java heraus, gestartet werden. Er lässt sich in fünf Teilaktionen unterteilen, die im Folgenden vorgestellt werden. Einen Überblick über den Ablauf gibt Abbildung 3.9.

Generierung des Vokabulars Der *InvertedFileParser* (*DataSource*) liest die Textdateien der Nutzdaten und extrahiert daraus alle Indexterme. Es besteht die Möglichkeit der Instanz einen Wortfilter zuzuweisen, um gleichzeitig eine Stoppwortentfernung vorzunehmen, wenn nicht bereits vorher geschehen. Außerdem könnte an dieser Stelle auch eine Wortstammreduktion stattfinden, die bei dieser Implementierung jedoch nicht vorgesehen ist. Alle gültigen Indexterme werden in ein Vokabular eingefügt, wobei Duplikate eliminiert werden. Die Unterscheidung von Groß- und Kleinschreibung kann dabei über einen Parameter aktiviert werden. Anschließend wird das Vokabular ausgelesen und in eine Textdatei auf die Festplatte geschrieben.

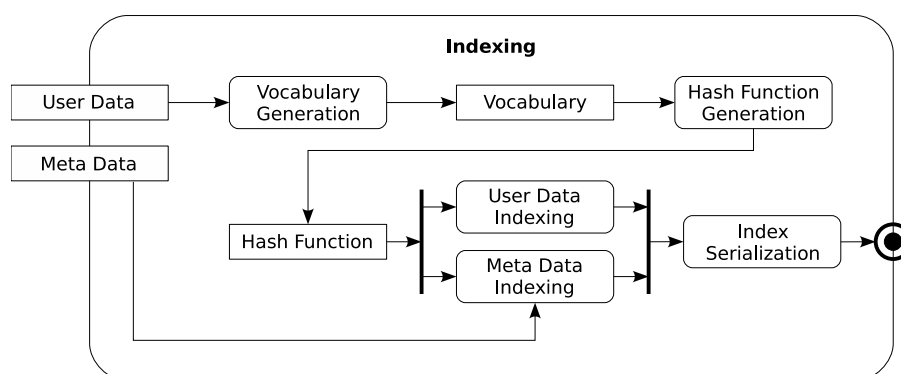


Abbildung 3.9: UML-Aktivität des Indizierungsprozesses.

Erzeugung der minimalen perfekten Hashfunktion Die Klasse *MPHashFunction* (*HashFunction*) liest die Datei des Vokabulars und konstruiert daraus eine minimale perfekte Hashfunktion (MPHF), wie sie in Abschnitt 3.2.5 noch vorgestellt wird. Dazu kapselt sie einen speziellen Algorithmus einer MPHF-Bibliothek. Die generierte Hashfunktion wird anschließend der Instanz des *InvertedFileParser* zugewiesen.

Indizierung der Nutzdaten Bei einem erneuten Parsing der Nutzdaten werden diesmal *DataItems* generiert, die an ein *DataStorage* weitergeleitet werden. Der Hashwert und die Prüfsumme eines *DataItems* werden durch die Hashfunktion im Parser berechnet. *DataItems* mit gleichem Hashwert werden zusammengeführt. Dieser Prozess ist in Abbildung 3.10 schematisch dargestellt.

Da dieser Vorgang bei großen Datenmengen nicht im Hauptspeicher erfolgen kann, implementiert die Klasse *ExternalDataStorage* ein externes Bucket-Sort. Dabei werden zunächst alle *DataItems* mittels einer einfachen Modulooperation auf temporäre Dateien (Buckets) aufgeteilt, wobei ihr Hashwert als Schlüssel dient. Zusammengehörige *DataItems*, d.h. diejenigen die den gleichen Hashwert besitzen, befinden sich dann im gleichen Bucket.

Im Hauptspeicher werden zwei Abbildungen $map_{h \rightarrow p}$ und $map_{h \rightarrow a}$ angelegt. $map_{h \rightarrow p}$ bildet Hashwerte auf Postlisten ab und dient temporär der Generierung der Postlisten für einen Bucket. $map_{h \rightarrow a}$ bildet Hashwerte auf physikalische Adressen der erzeugten und serialisierten Postlisten ab, und verbleibt permanent im Speicher. Beide Abbildungen können wegen des minimalen perfekten Hashings als einfache Arrays in der Größe des Vokabulars n realisiert werden.

Anschließend wird der erste Bucket in den Hauptspeicher geladen, um aus dessen *DataItems* Postlisten zu generieren. Dazu werden die enthaltenen *ByteArrays* in die entsprechende Postliste in $map_{h \rightarrow p}$ eingefügt. Wenn der Bucket komplett ausgelesen ist, werden alle nicht leeren Postlisten in Binärdateien auf die Festplatte gespeichert. Die physikalischen Adressen in Form von (Datei-Id, Offset)-Tupeln werden in $map_{h \rightarrow a}$ eingetragen. $map_{h \rightarrow p}$ wird geleert und der Vorgang für jeden weiteren Bucket wiederholt.

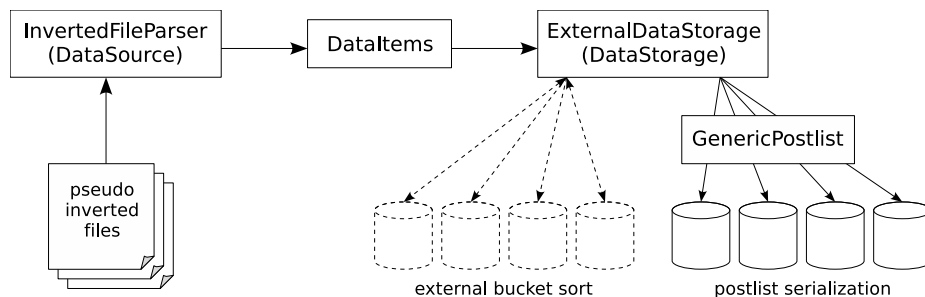


Abbildung 3.10: Indizierung der Nutzdaten mit dem *ExternalDataStorage*.

Bei Verwendung des *InternalDataStorage* dient $map_{h \rightarrow p}$ sowohl der Erzeugung als auch der permanenten Vorhaltung von Postlisten. Erst bei der Serialisierung des Index erfolgt deren externe Speicherung.

Indizierung der Metadaten Diese geschieht vollständig analog zur Indizierung der Nutzdaten über zwei weitere Instanzen einer *DataSource* und eines *DataStorages*. Zum Hashen der *DataItems* wird die bereits generierte Hashfunktion verwendet. Die *PostlistEntries* sind dabei immer vom Typ *StringStringTuple*.

Serialisierung des Index Bei der Serialisierung werden alle Parameter und Strukturen, die der erzeugte Index im Speicher hält, auf die Festplatte geschrieben. Dabei handelt es sich im Wesentlichen um die Hashfunktion und die jeweiligen Abbildungen der *DataStorages*. Der serialisierte Index kann anschließend beliebig oft geladen und zur Suche verwendet werden. Eine Abschätzung zum Speicherverbrauch zur Onlinezeit folgt in Abschnitt 3.2.6.

3.2.4 Suche im Index

Bevor der invertierter Index zur Suche verwendet werden kann, muss er einmalig von der Festplatte geladen werden. Bei dieser Deserialisierung werden alle internen Strukturen der *HashFunction* und der *DataStorages* im Hauptspeicher wiederhergestellt.

Eine Suche im Index liefert unter Angabe eines Suchworts die entsprechende Postliste zurückliefern. Je nachdem ob mit oder ohne Unterscheidung der Groß- und Kleinschreibung indiziert wurde, wird der übergebene Term zuerst in eine einheitliche Form konvertiert. Über die Hashfunktion wird dann der Hashwert und die Prüfsumme dieses Terms berechnet. Der Hashwert wird daraufhin zur Anfrage der Postliste an das *DataStorage* der Nutzdaten weitergeleitet. Über dessen interne Abbildung $map_{h \rightarrow a}$ bzw. $map_{h \rightarrow p}$ wird die zugehörige Postliste entweder von der Festplatte geladen oder direkt zurückgegeben. Analog dazu findet der Lookup der Metadaten statt.

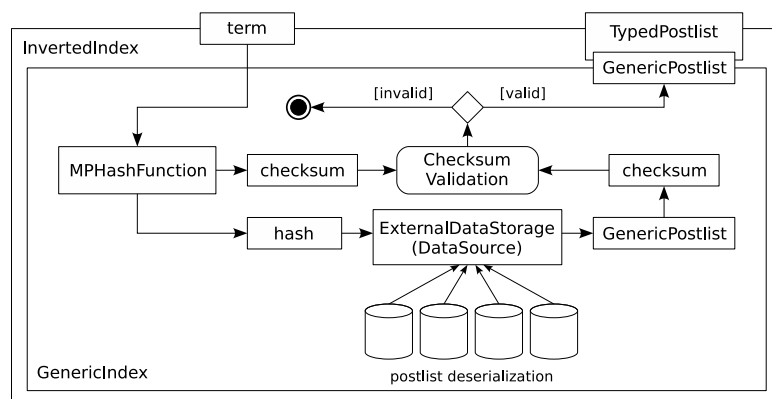


Abbildung 3.11: Lookup einer Postliste mit dem *ExternalDataStorage*.

Es kann der Fall eintreten, dass der Index mit einem Term anfragt wird, der nicht Teil des Vokabulars ist und für den dementsprechend auch keine Postliste im Index existiert. Die verwendete MPHF würde für diesen Term entweder einen gültigen Hashwert im Intervall $[0, n)$ berechnen oder einen Wert größer gleich n liefern. Letzteres widerspricht zwar der Definition einer Hashfunktion ist aber bei MPHF, die auf Zufallsgraphen basieren durchaus möglich. Der Grund hierfür liegt in der Berechnung des Grafen für ein minimales perfektes Hashing eines bestimmten Vokabulars und nicht für ein allgemeines Hashing beliebiger Terme. Handelt es sich dennoch um einen gültigen Hashwert kommt es zu einer Hashkollision und das *DataStorage* liefert erst einmal eine Postliste zurück. Um eine solche Kollision zu erkennen, wird die innerhalb der Postliste gespeicherte Prüfsumme mit der soeben berechneten Prüfsumme des Terms verglichen. Nur wenn diese übereinstimmen, wird die Postliste tatsächlich ausgeliefert. Die Wahrscheinlichkeit, dass ein unbekannter Term auch für die Prüfsumme eine Kollision erzeugt, beträgt bei der verwendeten 16-Bit-Prüfsumme $1/65521 = 0,000015262$ und ist somit vernachlässigbar gering.

3.2.5 Minimale perfekte Hashfunktion

Eine Hashfunktion h ist eine Abbildung, die jedem Schlüssel x aus einer Menge von Schlüsseln X eindeutig einen Ganzzahlenwert (Hashwert) zuordnet. Der Wertebereich von h liegt dabei im halboffenen Intervall $[0, m)$. Die Größe der Menge X wird mit n bezeichnet. Für Hashfunktionen gilt allgemein $n \gg m$. In den folgenden Betrachtungen wird die Menge X als Menge von Strings aufgefasst, wobei jeder String $x \in X$ aus einem Array von Zeichen besteht. Eine einfache Hashfunktion bildet die Modulooperation, bei der m mit einer Primzahl belegt ist:

$$h(x) = \sum_{i=0}^{|x|-1} x[i] \bmod m \quad (3.2)$$

Mit Hilfe einer Hashfunktion kann weiterhin eine Hashtabelle der Größe m definiert werden, deren Einträge (Slots) über h adressierbar sind. Jedem Schlüssel kann somit ein Slot zugewiesen werden. Bei Hashtabellen gilt allgemein $n < m$. Das Verhältnis $\alpha = n/m$ gibt den Loading-Faktor einer Hashtabelle an.

Eine Hashkollision tritt ein, wenn zwei voneinander verschiedene Schlüssel denselben Hashwert besitzen und somit auf denselben Slot verweisen. Die Wahrscheinlichkeit einer Kollision steigt exponentiell mit größer werdendem Loading-Faktor bei einer Zufallsverteilung der Hashwerte. Dieser Zusammenhang wird auch durch das Geburtstagsparadoxon³ veranschaulicht. Es existieren verschiedene Verfahren um Hashkollisionen zu behandeln; ein

³ Das Geburtstagsparadoxon stellt die Frage danach, wie viele Personen zusammengebracht werden müssen, damit die Wahrscheinlichkeit, dass zwei Personen am gleichen Tag (ohne Jahr) Geburtstag haben den Wert 0,5 übersteigt. Obwohl es 365 Tage im Jahr gibt, reichen bereits 23 Leute. Übertragen auf eine Hashtabelle mit 365 Slots ist demzufolge nach 23 Einfügeoperation eine Hashkollision mit einer Wahrscheinlichkeit von rund 0,5 zu erwarten.

Überblick dazu findet sich in [CLRS01]. Gute Hashfunktionen versuchen im Allgemeinen Kollisionen zu vermeiden.

Eine Hashfunktion wird als *perfekt* bezeichnet, wenn sie für keine zwei voneinander verschiedenen Schlüssel aus der Menge X eine Hashkollision erzeugt. Die Suche nach einer solchen Hashfunktion kann bei großem Loading-Faktor wegen der hohen Kollisionswahrscheinlichkeit sehr aufwändig sein. Eine Möglichkeit nach einer solchen Hashfunktion zu suchen, bietet Universal-Hashing. Hierbei wird versucht, die Parameter a und b einer gewöhnlichen Hashfunktion der Form 3.3 durch ein Trial-and-Error-Verfahren so zu wählen, dass keine Kollisionen auftreten.

$$h(x) = ((ax + b) \bmod p) \bmod m \quad (3.3)$$

Perfekte Hashfunktionen lassen sich weiterhin in *ordnungserhaltend* und *nicht-ordnungserhaltend* unterteilen. *Ordnungserhaltend* bedeutet, dass die relationale Ordnung der Schlüssel bzw. Terme in den Hashwerten erhalten bleibt.

$$x_1 < x_2 \Rightarrow h(x_1) < h(x_2) \quad (3.4)$$

Eine weitere Methode ist die Konstruktion einer *minimalen perfekten* Hashfunktion mit Hilfe eines Zufallsgraphen. *Minimal perfekt* bedeutet in diesem Zusammenhang, dass die Hashfunktion kollisionsfrei abbildet und $n = m$ ist. Diese Form der Hashfunktion ist optimal bezüglich Speicherverbrauch und Zugriffszeit, da weder leere Slots entstehen noch eine Kollisionsbehandlung nötig ist.

Für die Hashfunktion des invertierten Index wurde eine Implementierung dieser Art gewählt. Die *CMPH* (C Minimal Perfect Hashing Library) [dCRBB08b] bietet dazu die z.Z. modernsten und effizientesten Algorithmen. An dieser Stelle soll daher deren prinzipielle Funktionsweise am Beispiel des im Index angewandten BMZ-Algorithmus [dCRBB08a, BGZ, BKZ05] erläutert werden. Bei weiterem Interesse an der zugrundeliegende Grafentheorie sei auf die Literatur verwiesen (z.B. [CLRS01]). In den folgenden Betrachtungen bezeichnet x einen Term aus dem Vokabular X des Index. Die MPH berechnen sich nach der Form:

$$h(x) = g(h_1(x)) + g(h_2(x)) \quad (3.5)$$

Die Funktionen h_1 und h_2 entsprechen herkömmlichen Hashfunktionen, die alle Terme auf das Intervall $[0, m)$ mit $m = c \cdot n$ und möglichen Kollisionen verteilen (vgl. 3.2). Der Wert für m entspricht der Knotenanzahl in einem Zufallsgraphen, den es zu konstruieren gilt. In [BGZ] und [BKZ05] wird gezeigt, dass der optimale Wert für den Parameter c bei 1,15 liegt. Um unterschiedliche Abbildungen zu erhalten, enthält jede der beiden Funktionen zusätzlich einen ausreichend langen Zufallsvektor, der jedes Zeichen des Terms gewichtet.

$$h_1(x) = \left(\sum_{i=0}^{|x|-1} x[i] \cdot w_1[i] \right) \bmod m \quad (3.6)$$

$$h_2(x) = \left(\sum_{i=0}^{|x|-1} x[i] \cdot w_2[i] \right) \bmod m \quad (3.7)$$

Für jeden Term $x \in X$ wird zuerst das Paar $\{h_1(x), h_2(x)\}$ berechnet. Dabei muss für jedes Paar $h_1(x) \neq h_2(x)$ gelten und keines darf doppelt auftreten. Andernfalls müssen die Zufallsvektoren w_1 und w_2 neu gewählt werden.

Im nächsten Schritt muss eine Abbildung g bestimmt werden, die die berechneten Werte von h_1 und h_2 aus dem Intervall $[0, m)$ auf das Intervall $[0, n)$ abbildet und zwar auf die Weise, dass bei Anwendung der Formel 3.5 tatsächlich ein minimales perfektes Hashing für die Menge X entsteht. Die Abbildung g wird dazu als statischer Vektor der Länge m realisiert, dessen Elemente über das Anlegen und Gewichten eines Grafen bestimmt werden. Hierin liegt der entscheidende Schritt bei der Konstruktion einer MPHf, bei dem sich die verschiedenen Algorithmen hauptsächlich unterscheiden. Zur Erzeugung von ordnungserhaltenden Hashfunktionen werden azyklische Grafen und für nicht-ordnungserhaltende Hashfunktionen zyklische Grafen verwendet.

Der BMZ-Algorithmus erzeugt eine nicht-ordnungserhaltende Funktion mit Hilfe eines Grafen $G = (V, E)$ mit $|V| = m$ und $|E| = n$. Die Kantenmenge $E = \{\{h_1(x), h_2(x)\} : x \in X\}$ enthält alle Kanten, deren Knoten für alle $x \in X$ mit den Hashfunktionen h_1 und h_2 bestimmt wurden. Jede Kante repräsentiert also einen Term aus dem Vokabular. Die Menge der Knoten wird durchnummeriert. Abbildung 3.12 zeigt beispielhaft eine Tabelle mit einem Vokabular von neun Termen mit ihren Hashwerten h_1 und h_2 sowie den entsprechenden Zufallsgraphen.

Term x	$h_1(x)$	$h_2(x)$
index	5	9
indexed	9	1
indexer	6	4
indexical	4	3
indexically	1	4
indexing	6	1
indexless	8	5
indexlessness	9	8
indexterity	3	10

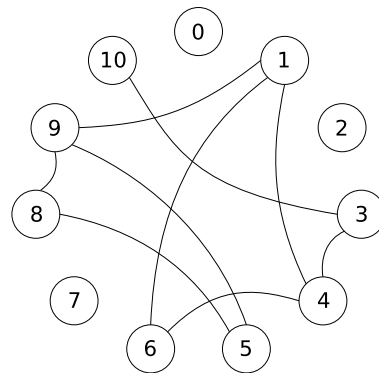


Abbildung 3.12: Zufallsgraf zu einem Vokabular.

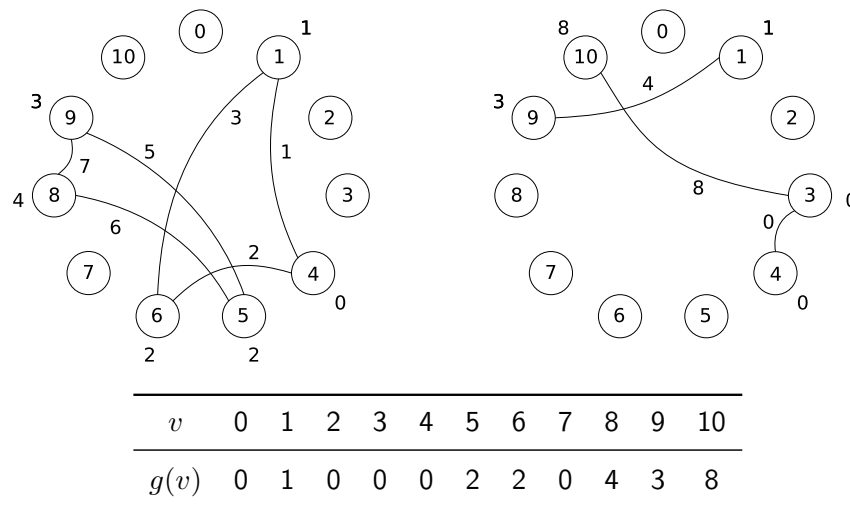


Abbildung 3.13: Zerlegung und Gewichtung des Zufallsgraphen.

Alle Knoten und Kanten erhalten ein ganzzahliges Gewicht, wobei sich das Gewicht einer Kante aus der Addition der adjazenten Knotengewichte ergibt. Gesucht wird eine bestimmte Gewichtung für V , so dass die Gewichtung für E einem minimalen perfekten Hashing für X entspricht. Der Graph G wird dazu zunächst anhand seiner Zyklen in zwei Subgraphen G_{zykl} und G_{azykl} zerlegt, die nach einem Algorithmus gewichtet werden. Die Gewichtung der Knoten in G_{zykl} folgt dabei einer Greedy-Strategie, um die durch Zyklen bedingte Mehrfachzuweisung von Gewichten aufzulösen. Die Gewichtung der Knoten in G_{azykl} ist dagegen unkritisch. Der genaue Prozess wird in [BKZ05] ausführlich beschrieben.

Abbildung 3.13 zeigt die Zerlegung in G_{zykl} (links) und G_{azykl} (rechts) mit der ermittelten Gewichtung. Alle Knotengewichte werden in den Vektor g eingetragen. Unter Anwendung der Formel 3.5 kann schließlich das perfekte minimale Hashing berechnet werden, das in Abbildung 3.14 für das Beispieldvokabular zusammenfassend dargestellt ist.

Term x	$h_1(x)$	$h_2(x)$	$h(x)$
index	5	9	5
indexed	9	1	4
indexer	6	4	2
indexical	4	3	0
indexically	1	4	1
indexing	6	1	3
indexless	8	5	6
indexlessness	9	8	7
indexterity	3	10	8

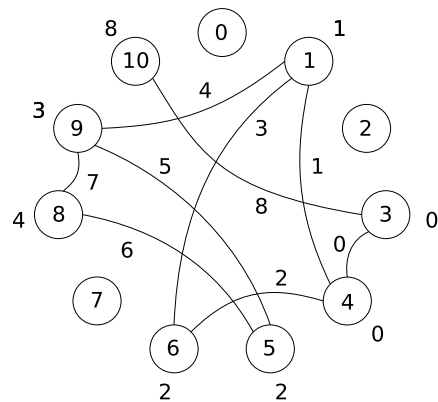


Abbildung 3.14: Minimales perfektes Hashing durch gewichteten Zufallsgraphen.

3.2.6 Abschätzung zum Speicherverbrauch

Dieser Abschnitt gibt eine Abschätzung über den Speicherverbrauch des Index zur Onlinezeit. Mit der Deserialisierung des Index werden neben wenigen Parametern die MPHF sowie die *DataStorages* der Nutz- und Metadaten in den Hauptspeicher geladen. Es wird im Folgenden davon ausgegangen, dass für beide *DataStorages* Instanzen vom Typ *ExternalDataStorage* verwendet werden. Bei Verwendung der Klasse *InternalDataStorage* entspricht der Speicherverbrauch der geladenen Datenmenge, die in der Regel so groß ist, dass alle weiteren Strukturen kaum ins Gewicht fallen.

Die abhängige Variable ist bei allen Komponenten die Größe des Vokabulars n . Bei der verwendeten MPHF bestimmt n die Größe der Abbildung g , die die Knotengewichte des Zufallsgraphen speichert. Die Anzahl der Knoten berechnet sich durch $|V| = c \cdot n$, wobei c mit 1,15 belegt ist. Zur Repräsentation der Hashfunktion sind lediglich die Parameter der beiden gewöhnlichen Hashfunktionen h_1 und h_2 sowie die Abbildung g nötig. Da g als einfaches Array von 32-Bit-Integerwerten realisiert ist und h_1 und h_2 von nur sehr geringer konstanter Größe sind, ergibt sich der Speicherverbrauch der MPHF wie folgt:

$$mem_{mphf} = 4 \cdot 1,15 \cdot n \quad [Byte] \quad (3.8)$$

Für eine *ExternalDataStorage* ergibt sich der Speicherverbrauch hauptsächlich aus der Abbildung $map_{h \rightarrow a}$, die Hashwerte auf (Datei-Id, Offset)-Tupel abbildet. Aus Effizienzgründen ist diese Abbildung in Form von zwei Arrays der Größe n realisiert. Das erste enthält die Datei-Ids als 8-Bit-Integer, das zweite die Offsets als 32-Bit-Integer. Ein drittes Array, das die maximal 256 File-Handler der geöffneten Dateien verwaltet, kann vernachlässigt werden.

$$mem_{storage} = (8 + 32) \cdot n \quad [Byte] \quad (3.9)$$

Bei einer Indizierung mit Metadaten ist dieser Wert entsprechend zu verdoppeln. Als untere Schranke für den Speicherverbrauch ergibt sich somit:

$$mem_{total} = mem_{mphf} + 2 \cdot mem_{storage} = (4,6 + 80) \cdot n \quad [Byte] \quad (3.10)$$

Für die Indizierung der Google-N-Gramme mit Metadaten und einem Vokabular von 2.236.483 Indextermen für den NETSPEAK-Dienst beträgt dieser Wert:

$$mem_{total} = (4,6 + 80) \times 2.236.483 = 180,4 \quad [Megabyte] \quad (3.11)$$

3.2.7 Anbindung an Java

Der NETSPEAK-Dienst wurde in der Programmiersprache Java entwickelt. Java bietet mit dem Java-Development-Kit (JDK) eine Bibliothek mit einer breiten Unterstützung verschiedener Web-Technologien. Sie ist deshalb die erste Wahl bei der Realisierung umfangreicher Web-Applikationen. Der Index ist hingegen in C++ implementiert. Für C++ sprechen im Allgemeinen eine schnellere Laufzeit und eine bessere Unterstützung hardwarenaher Operationen, wie Dateizugriffe und Speichermanagement. Um die Indeximplementierung auch für Java-Anwendungen zur Verfügung zu stellen, wurde eine Schnittstelle mit Hilfe des Java-Native-Interface (JNI) geschaffen.

Das JNI ist eine Java-Technologie, die es ermöglicht innerhalb der virtuellen Maschine von Java (JVM) nativen Code in C, C++ oder Assembler auszuführen. Der native Code muss dazu als dynamisch linkbare Bibliothek kompiliert und eingebunden werden. Bei Linux-/Unix-artigen Betriebssystemen handelt es sich dabei um Dateien mit der Endung *.so* (Shared-Object) und bei Windows um *.dll* (Dynamic-Link-Library). Im Java-Code müssen Methoden, die JNI nutzen, als *native* deklariert werden, um der JVM zu signalisieren, dass sich deren Implementierung in der eingebundenen nativen Bibliothek befindet.

Bei der Arbeit mit dem JNI ist es sehr wichtig zu verstehen, dass es sich dabei um eine funktional-orientierte Technologie handelt. Das bedeutet, dass ein als *native* deklariertes Methodenaufruf in Java immer einem nativen Funktionsaufruf in C++ entspricht. Es ist zwar möglich zwischen Java und C++ Eingabe- und Rückgabeparameter auszutauschen, doch es ist nicht vorgesehen auf der Seite von C++ Daten oder Objekte persistent zu halten. Die in einer nativen Funktion erzeugten Objekte werden entweder bei deren Terminierung gelöscht oder verbleiben bei dynamischer Allokierung auf dem Heap. Letzterer Fall würde ein Speicherleck verursachen, da kein Pointer auf das Objekt gespeichert kann, um es wieder freizugeben. Auf dieses Problem wird später noch einmal eingegangen.

Zunächst soll erläutert werden, wie die Übertragung von Postlisten aus dem Index von C++ nach Java rein logisch stattfindet. Eine Anforderung beim Design der Schnittstelle war, dass Postlisten mit beliebigen *PostlistEntries* übertragen werden können ohne dabei das JNI des Index jedes Mal neu anpassen zu müssen. Die Java-Anbindung konnte also nicht auf der Benutzerebene stattfinden, sondern musste im Uninterpreted-User-Data-Layer ansetzen, wo Elemente einer Postliste noch durch generische *ByteArrays* repräsentiert werden (vgl. 3.6). Der Index wurde hierfür ab dieser Ebene aufwärts in Java reimplementiert. Die C++-Klassen *GenericPostlist*, *TypedPostlist*, *PostlistEntry* und *InvertedIndex* existieren hier ebenso mit gleicher Funktionalität. Der tatsächliche Austausch der Daten findet nun von einer *GenericPostlist* in C++ zu einer *GenericPostlist* in Java statt, indem jedes *ByteArray* in ein Java *byte[]* kopiert wird. Diese sowie alle weiteren Methoden zum Erzeugen und Laden des Index werden über JNI-Aufrufe an deren entsprechende native Äquivalente delegiert.

Um auf das Problem des gedächtnislosen JNI zurückzukommen, sei an die sehr großen Postlisten erinnert, die nicht mit einmal in den Speicher geladen werden können, sondern abschnittsweise ausgelesen werden müssen. Eine solche Postliste muss nach dem nativen Lookup auf Seite von C++ und teilweisem Kopieren ihrer Elemente im Hauptspeicher verbleiben, um ein späteres Swapping der Java-Postliste zu ermöglichen. Ähnliches gilt natürlich für das Indexobjekt selbst.

Die Lösung besteht darin, bei der Erzeugung eines C++-Objektes in einer nativen Funktion, eine Art Referenz darauf auf die Java-Seite zu kopieren. Diese Referenz soll später dazu dienen auf dasselbe Objekt noch einmal zugreifen zu können. Ein erster Ansatz hierfür ist, als Referenz den Pointer des dynamisch allokierten Objektes zu benutzen. Da Java natürlich keine C++-Pointer verarbeiten kann, muss dieser für den Austausch in einen Integer gecastet werden. Bei jedem JNI-Aufruf, etwa zum Löschen einer nativen Postliste, wird deren Pointer als Integer mitgeführt und vor der Wiederbenutzung innerhalb der nativen Funktion in einen Pointer zurückgecastet. Leider funktioniert dieser Ansatz nur auf 32-Bit-Maschinen, bei denen ein Pointer und ein Integer 32 Bit groß sind. Bei gleichem Vorgehen auf einem 64-Bit-System verweigert der C++-Compiler seinen Dienst, da er einen 64-Bit-Pointer nicht auf einen 32-Bit-Integer casten kann. Die Umstellung von 32-Bit-Integer auf 64-Bit-Integer als Referenz führt ebenfalls nicht zum Erfolg, da das Casting in die andere Richtung von 64-Bit-Integer auf einen 32-Bit-Pointer unter 32-Bit-Systemen nicht möglich ist.

Ein zweiter Ansatz, der auch so implementiert wurde, ist die Einrichtung eines persistenten Objekt-Pools auf Seite von C++. Dieser Pool kann jederzeit aus einer nativen Funktion heraus angesprochen werden, um Objekte darin abzulegen oder herauszunehmen. Für jedes neu eingefügte Objekt liefert der Objekt-Pool einen eindeutigen 32-Bit-Integer als Objekt-Id bzw. Referenz. Unter Angabe dieser Objekt-Id kann später auf dasselbe native Objekt wieder zugegriffen werden. Dieser Ansatz ist plattformunabhängig, da die Größe dieser Objekt-Id, im Gegensatz zu einem Pointer, auf jeder Architektur konstant ist. Die beschriebene Funktionalität wird in der Indeximplementierung durch die Klasse *JNIObjectPool* realisiert.

4 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde das Informationssystem NETSPEAK vorgestellt. NETSPEAK ist ein neuartiger Web-Dienst, der das Suchen nach geläufigen Phrasen in der englischen Sprache ermöglicht, um damit Autoren beim Verfassen englischsprachiger Texte zu unterstützen. Dazu dient NETSPEAK das World Wide Web als Quelle für zeitgemäße und richtige Sprache. Dabei werden die Annahmen gemacht, dass geläufige Formulierungen insgesamt häufiger im Web auftreten als weniger geläufige oder sogar falsche Sprachkonstrukte.

NETSPEAK stellt eine einfache Anfragesprache bereit, um nach solchen kurzen Textphrasen suchen zu lassen. Dafür unterstützt es Wildcard-Queries, bei denen für unbekannte Worte Platzhalter eingefügt werden können. Das Ergebnis einer Suche ist eine Liste mit gültigen Formulierungen in englischer Sprache, in denen diese Platzhalter durch sinnvolle Worte ersetzt sind. Eine Sortierung der gefundenen Phrasen anhand ihrer Häufigkeit im Web ermöglicht anschließend eine Bewertung hinsichtlich ihrer Gebräuchlichkeit.

NETSPEAK verwendet als Datenbasis eine Kollektion von rund einer Milliarde 5-Grammen, die aus allen durch Google indizierten englischsprachigen Web-Seiten erstellt wurde. Diese Kollektion wurde mit dem Verfahren der invertierten Liste indiziert, um für eine Anfrage schnell nach passenden Phrasen suchen zu können. Die Implementierung dieses speziellen Index wurde ebenfalls in dieser Arbeit vorgestellt. Es zeigte sich, dass dieser durch Verwendung einer minimalen perfekten Hashfunktion optimale Ergebnisse in Bezug auf Speicherplatz und Lookup-Performanz bietet.

NETSPEAK ist z.Z. als Prototyp implementiert und liefert für einzelne Anfragen bereits sehr gute Ergebnisse mit kurzen Antwortzeiten. Die Weiterentwicklung wird nun darin bestehen, die Qualität der Suchergebnisse weiter zu verbessern. Hierbei müssen insbesondere die angesprochenen Probleme bei der Rechtschreibkorrektur und der Synonymsuche für flektierte Worte gelöst werden. Weiterhin ist vorgesehen, NETSPEAK zu einem Web-Dienst auszubauen, der in der Lage ist eine große Anzahl von parallelen Suchanfragen zu beantworten. Dazu gilt es neben der Implementierung der in Abschnitt 2.7 diskutierten Optimierungsmöglichkeiten einen Load-Balancing-Cluster aufzubauen. Durch die redundante Vorhaltung der Indexdatenstruktur und der Google-N-Gramm-Kollektion auf diesem Cluster wird die Performanz von NETSPEAK erheblich gesteigert werden können. Wenn diese Maßnahmen abgeschlossen sind, wird es schließlich spannend zu beobachten sein, ob NETSPEAK als nützliche Suchmaschine angenommen wird.

Literaturverzeichnis

- [BGZ] Fabiano C. Botelho, David M. Gomez, and Nivio Ziviani.
A New Algorithm for Constructing Minimal Perfect Hash Functions.
Technical Report TR004/04,
Department of Computer Science, Federal University of Minas Gerais.
- [BKZ05] Fabiano C. Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani.
A Practical Minimal Perfect Hashing Method. In *Lecture Notes in Computer Science*, pages 488–500. Springer Verlag, May 2005.
<http://cmph.sourceforge.net/papers/wea05.pdf>.
- [BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto.
Modern Information Retrieval. Addison Wesley Longman, May 1999.
<http://www.sims.berkeley.edu/~hearst/irbook>.
- [CLRS01] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein.
Introduction to Algorithms. MIT Press, second edition, 2001.
- [dCRBB08a] Davi de Castro Reis, Djamel Belazzougui, and Fabiano C. Botelho.
BMZ Algorithm, 2008.
<http://cmph.sourceforge.net/bmz.html>.
- [dCRBB08b] Davi de Castro Reis, Djamel Belazzougui, and Fabiano C. Botelho.
CMPH - C Minimal Perfect Hashing Library, 2008.
<http://cmph.sourceforge.net/index.html>.
- [Fuh96] Norbert Fuhr.
Fachgruppe Information Retrieval. *Gesellschaft für Informatik*, 1996.
http://www.uni-hildesheim.de/fgir/index.php?option=com_content&task=view&id=14&Itemid=41.
- [IL06] Google Inc. and Linguistic Data Consortium (LDC).
Web 1T 5-gram Version 1. 2006.
Catalog Number LDC2006T13, ISBN 1-58563-397-6.
- [Inc06] Silicon Graphics Inc.
Standard Template Library Programmer's Guide, 1993-2006.
<http://www.sgi.com/tech/stl/sort.html>.

- [Kuh90] Rainer Kuhlen.
Zum Stand pragmatischer Forschung in der Informationswissenschaft. In *Pragmatische Aspekte beim Entwurf und Betrieb von Informationssystemen. Proceedings des 1. Internationalen Symposiums für Informationswissenschaft*, pages 13–18. Universitätsverlag Konstanz, 1990.
- [SC06] Benno Stein and Daniel Curatolo.
Phonetic Spelling and Heuristic Search. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *17th European Conference on Artificial Intelligence (ECAI 06)*. IOS Press, August 2006.
- [Sed93] Robert Sedgewick.
Algorithmen in C. Addison Wesley, 1993. 3. Nachdruck 1998.
- [Slo04] Joseph D. Sloan.
High Performance Linux Clusters. O’Reilly Media, 2004.
- [Ste07] Benno Stein.
Grundlagen des Information Retrieval: Indexterme.
Vorlesungsskripte, Kollektion Information Retrieval, 2005-2007.
<http://www.uni-weimar.de/cms/medien/webis/teaching/lecture-notes.html#information-retrieval>.
- [vR79] Keith van Rijsbergen.
Information Retrieval. Butterworths, second edition, 1979.
<http://www.dcs.gla.ac.uk/Keith/Preface.html>.
- [Wik08a] Wikipedia. *Amdahlsches Gesetz*. 21/06/2008.
http://de.wikipedia.org/wiki/Amdahls_Gesetz.
- [Wik08b] Wikipedia. *Google*. 21/06/2008.
<http://de.wikipedia.org/wiki/Google>.
- [WMB94] Ian H. Witten, Alistair Moffat, and Timothy C. Bell.
Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, second edition, March 1994.

A Netspeak Service

Dieser Anhang befindet sich auf der beiliegenden CD im Verzeichnis `/netspeak`.

A.1 Eclipse Projekt (Java)

Die Eclipse-Projektdateien bestehen aus den beiden Komponenten `service` und `ui`. In `service` befindet sich der Quellcode zum Retrieval-Prozess und `ui` enthält alle nötigen Dateien für das User-Interface. Das Verzeichnis `conf` beinhaltet die Konfigurationsdatei und in `build` ist das XML-Build-File zur Erzeugung eines WAR-Archivs für den Tomcat-Servlet-Container abgelegt.

A.2 Dokumentation (Javadoc)

Die Dokumentation aller beteiligten Java-Klassen findet sich unter `doc`.

B Inverted Index

Dieser Anhang befindet sich auf der beiliegenden CD im Verzeichnis `/invertedindex`.

B.1 Implementation

B.1.1 Codeblocks Projekt (C++)

Der Quellcode des invertierten Index in C++ befindet sich im Verzeichnis `src/cpp`. Hier finden sich außerdem die dazugehörigen Projektdateien der Code::Blocks IDE für Linux (32- und 64-Bit-System) und Windows (32-Bit-System).

B.1.2 Eclipse Projekt (Java)

Der Quellcode des invertierten Index in Java befindet sich im Verzeichnis `src/java`. Die Verzeichnisstruktur unter `/invertedindex` kann direkt als Eclipse-Projekt eingebunden werden.

B.2 Dokumentation

B.2.1 C++ (Doxygen)

Die Dokumentation aller beteiligten C++-Klassen findet sich unter `doc/cpp`.

B.2.2 Java (Javadoc)

Die Dokumentation aller beteiligten Java-Klassen findet sich unter `doc/java`.