Universität Trier — Fachbereich VI — Informatik

Diplomarbeit

Entwicklung einer Suchmaschine für bibliographische Daten am Beispiel des DBLP

Thorsten Thielen

August 2007

Begutachtet von Prof. Dr. Bernd Walter
und Prof. Dr. Peter Sturm

Betreut von Dr. Michael Ley
und Dr. Patrick Reuther

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Einführung

Motivation und Zielsetzung

Beim DBLP („Digital Bibliography & Library Project“) handelt es sich um eine umfangreiche, frei zugängliche Sammlung bibliographischer Informationen zu allen Arten wissenschaftlicher Publikationen aus dem Bereich der Informatik.

Wie bei den meisten großen Datenbeständen, so bedarf es auch hier vor allem effizienter, einfach zu bedienender und ausreichend mächtiger Werkzeuge zum Auffinden von gewünschten Informationen um eine sinnvolle Nutzung dieser Daten zu ermöglichen. Die bisher für das DBLP verfügbaren Suchmöglichkeiten werden diesem Anspruch jedoch nur bedingt gerecht.

Aus diesem Grund wurden im Rahmen dieser Diplomarbeit die bestehenden Suchwerkzeuge des DBLP um eine spezialisierte Suchmaschine ergänzt. Diese stellt, neben der bisherigen einfachen Volltextsuche nach Autor und Titel und der erweiterten „Advanced“–Suche, eine mächtigere, auf die speziellen Bedürfnisse und Inhalte des DBLP–Datenbestandes angepasste Suche bereit.

Gliederung der Arbeit

Am Anfang der Arbeit werden in *Kapitel 1* die bisher bestehenden Browsing- und insb. Suchalternativen analysiert und auf ihre Mängel und Schwachstellen untersucht. Des weiteren werden eine Reihe von zusätzlichen Suchtypen vorgestellt, die für den DBLP–Datenbestand sinnvolle Abfragemöglichkeiten repräsentieren. Auf der Basis dieser Zusammenstellungen werden dann die genauen Zielvorgaben für die neue Suchanwendung spezifiziert.

Die Analyse des bestehenden DBLP–Datenbestandes, als Grundlage für die Planung, erfolgt in *Kapitel 2*. Untersucht werden die Dateiformate in denen die Daten abgegriffen werden können, inklusive der jeweiligen Vor- und Nachteile; dann der syntaktische Aufbau der Datei `dblp.xml` sowie die genaue Art, der Aufbau und die Menge der enthaltenen Daten. Im Zuge dessen wird ebenfalls kurz das Programm `dblpanalyze` vorgestellt, mit die statistischen Daten ermittelt wurden.

Als weitere, vorbereitende Überlegungen werden in *Kapitel 3* einige allgemeine, frei verfügbare Suchmaschinen vorgestellt und abgewogen, welche davon als Basis der neuen DBLP–Suche am geeignetsten ist. Ebenso werden einige Erwägungen bzgl. der zur verwendenden Programmiersprache und der Gestaltung der Nutzeroberfläche angestellt.

Die Datenstruktur des Invertierten Index mit der dahinterstehenden Motivation, seinem Aufbau und die bei der Generierung relevanten Schritte und Aspekte werden in *Kapitel 4* erläutert. Anhand des Datei–Formats und der an der Erstellung beteiligten Klassen der verwendeten Lucene–Suchmaschine wird eine konkrete Implementierung eines solchen Index vorgestellt. Zum Aufbau des Suchindex für die neue DBLP–Suche dient das Programm `dblp2index`; neben Informationen zur Bedienung und Performance wird dargelegt, welche genauen Daten des DBLP–Datenbestandes dort in welche Weise importiert werden und welche Gründe dahinter stehen.

Die allgemeine Vorgehensweise bei verschiedenen Suchabfrage–Methoden sowie die Verflechtungen mit dem Index–Format werden schließlich in *Kapitel 5* kurz behandelt. Wiederum anhand von Lucene werden dann der konkrete Ablauf und die beteiligten Klassen einer spezifischen Implementierung angerissen. Schlussendlich wird auf das Programm `dblpsearch` eingegangen, welches die eigentliche Suchanwendung darstellt; Erläuterungen zur Bedienung und ein kurzer Vergleich mit den bisherigen Suchen schließen die Arbeit ab.

Im Anhang finden sich ergänzend eine Übersicht über die Daten des Test- und Entwicklungssystems, welches bei dieser Arbeit zum Einsatz kam. Des weiteren wird dort die ebenfalls im Zuge dieser Arbeit entwickelte Bibliothek `libdblp` kurz vorgestellt, auf die sich viele der Programme zur neuen DBLP–Suche stützen.

Anmerkungen

Eine elektronische Ausgabe dieser Diplomarbeit kann im WWW unter der URL `http://diplomarbeit.c2226.de/` heruntergeladen werden. Die im Zuge dieser Arbeit entwickelten Programme bzw. Bibliotheken sind auf der beiliegenden CD bzw. ebenfalls unter der angegebenen URL verfügbar.

Die Angabe von Speichergrößen in dieser Arbeit folgt dem International Standard IEC 60027-2 [NIST00]. Dabei bezeichnet „MiB“ die Einheit „mebibyte“; 1 MiB $= 2^{20}$ Bytes $= 1.048.576$ Bytes; „GiB“ steht für „gibibyte“; 1 GiB $= 2^{30}$ Bytes $= 1.073.741.824$ Bytes.

Kapitel 1

Motivation und Zielsetzung

Die erste Maßnahme bei der Bearbeitung eines Projektes, insb. auch im Bereich der Softwareentwicklung, sollte es sein, sich einen Überblick über die Voraussetzungen und die gestellte Aufgabe zu verschaffen. Die dabei gewonnenen Erkenntnisse bestimmen dann die weitere Herangehensweise.

Das allgemeine Ziel dieser Arbeit ist die Planung und Implementierung einer neuen Suche für das DBLP–Projekt. In diesem Kapitel werden daher das DBLP und die bisher dafür verfügbaren Browsing- und Suchmöglichkeiten kurz vorgestellt.

Um die notwendigen Schritte für die Entwicklung der Anwendung genauer einschätzen und spezifizieren zu können, werden die vorhandenen Schwächen dieser Suchmöglichkeiten untersucht. Des weiteren werden Überlegungen angestellt, welche neuen Features bzw. Erweiterungen in diesem Bereich sinnvoll einzusetzen wären.

Aufgrund der erhaltenen Informationen wird schließlich die genaue Zielsetzung dieser Arbeit mit den an die neue Applikation zu stellenden Anforderungen formuliert.

1.1 Das DBLP

Das DBLP ist eine umfangreiche, frei zugängliche Sammlung bibliographischer Informationen zu allen Arten wissenschaftlicher Publikationen aus dem Bereich der Informatik. Aktuell (Juli 2007) beinhaltet das DBLP mehr als 920.000 Einträge und wird ständig erweitert.

Das DBLP wird an der Universität Trier gehosted[1], daneben gibt es verschie-

[1] siehe http://dblp.uni-trier.de/

dene weitere Mirror–Sites. Betreut wird es von Dr. Michael Ley[2], der für diese Arbeit im Jahre 1997 mit dem ACM SIGMOD Service Award sowie dem VLDB Endowment Special Recognition Award ausgezeichnet wurde.

Am Anfang seiner Entstehung in den 1990er Jahren konzentrierte sich das DBLP auf Veröffentlichungen aus dem Bereich Datenbanken und Logikprogrammierung, woraus auch die ursprüngliche Bezeichnung „DataBase and Logic Programming“ resultiert. In jüngerer Zeit erweiterte sich die Bandbreite auch auf andere Gebiete der Informatik, so dass die Abkürzung heute als „Digital Bibliography & Library Project“ gelesen werden sollte.

1.2 Vorhandene Browsing- und Suchmöglichkeiten des DBLP

Grundlage für eine effiziente Nutzung großer Datenbestände wie dem des DBLP ist die Möglichkeit, relevante Informationen schnell, zielsicher und effizient auffinden zu können.[3] Neben der Möglichkeit des manuellen Suchens mittels Browsing (sofern eine geeignete Visualisierung/Oberfläche verfügbar ist) spielt hier insb. die automatisierte Suche unter Zuhilfenahme geeigneter Suchmaschinen die wichtigste Rolle.

Für das DBLP existieren z.Zt. zwei „offizielle“ Zugriffsmöglichkeiten:

- Die Weboberfläche[4] ermöglicht einen Online–Zugriff auf die aktuellsten DBLP–Daten über das WWW;
- der DBLP–Browser[5] erlaubt die Suche und Betrachtung der DBLP–Daten offline.

Bei beiden Oberflächen kann sowohl mittels Browsing- als auch über unterschiedliche Suchen auf die Daten zugegriffen werden.

1.2.1 Browsing mittels Weboberfläche

Die DBLP–Website ist weitgehend hierarchisch aufgebaut:

Von der Startseite aus referenziert werden die Liste aller im Datenbestand erfassten Konferenzen („Computer Science Conferences & Workshops“) sowie die

[2] siehe http://www.informatik.uni-trier.de/~ley/addr.html
[3] vgl. [CCG+72], „Olympic Hide–and–Seek Final“
[4] siehe http://dblp.uni-trier.de/
[5] siehe http://dbis.uni-trier.de/DBL-Browser/

Liste aller erfassten Zeitschriften („Computer Science Journals“). Diese beiden Indizes führen, alphabetisch nach dem Namen (bzw. bei den Konferenzen meist nach der Abkürzung desselben) geordnet, die erfassten Konferenzen bzw. Zeitschriften auf.

Jeder Eintrag in diesen Listen verweist auf eine Seite mit Informationen zum „publication stream“. Dabei handelt es sich um eine chronologische Auflistung der Sitzungsberichte (Proceedings) der jeweiligen Konferenz bzw. der Ausgaben (Volumes) der jeweiligen Zeitschrift, teilweise ergänzt durch weitere bibliographische Informationen, Hinweise auf zukünftige Veranstaltungen, Links auf die Webseiten der Herausgeber und Ähnliches.

Die einzelnen Einträge für die Proceedings bzw. Volumes wiederum führen dann zu den Inhaltsverzeichnissen (TOC, Table Of Contents) der entsprechenden Publikation. Dort findet sich dann eine Auflistung der einzelnen Beiträge bzw. Artikel mit Angaben zu Autor bzw. Autoren, Links auf elektronische Ausgaben (falls verfügbar), etc.

Über die Autorennamen verlinkt sind die Autorenseiten. Auf diesen finden sich, chronologisch geordnet, Informationen zu allen Publikationen des jeweiligen Autors. Des weiteren findet sich hier auch eine Liste aller Co–Autoren, welche bisher zusammen mit dem besagten Autor publiziert haben.

Die Autorenseiten sind alternativ auch direkt über den Autorenindex erreichbar. Dieser listet (aus Gründen der Übersichtlichkeit in zwei Stufen) alle erfassten Autoren in alphabetischer Reihenfolge auf. Der Autorenindex stellt neben dem Weg über die Konferenzen bzw. Zeitschriften eine zweite, wichtige Zugriffsmöglichkeit auf die Informationen des DBLP dar.

Für jede Publikation existiert eine eigene Seite, welche die (wichtigsten) Informationen des entsprechenden Datensatzes im BibTEX[6]–Format bereitstellt. Auf diese Weise ist es sehr einfach möglich, einen entsprechenden Eintrag z.B. im Literaturverzeichnis wissenschaftlicher Arbeiten zu erzeugen.

Schließlich gibt es noch die Liste der Themengebiete („Subjects Page“). Diese fasst die Konferenzen und Zeitschriften nach den dort (hauptsächlich) behandelten Themen zusammen und erlaubt auf diese Weise einen nach thematischen Gesichtspunkten orientierten Zugriff auf den Datenbestand des DBLP.

1.2.2 Autoren- bzw. Titelsuche mittels Weboberfläche

Neben diesem „manuellen“ Zugriff auf die DBLP–Daten gibt es die Möglichkeit, direkt nach Autoren(namen) bzw. Publikationstiteln zu suchen. Durch einfache Eingabe eines Suchbegriffes erlauben es diese Suchen, diejenigen Autoren bzw.

[6] siehe z.B. http://en.wikipedia.org/wiki/BibTeX

Publikationen zu finden, in deren Namen bzw. Titel die entsprechende Zeichenkette vorkommt.

Abbildung 1.1: Autorensuche auf der DBLP–Weboberfläche

Bei beiden Suchen handelt es sich um einfach Substring–Suchen mit nur sehr rudimentären Steuerungsmöglichkeiten:

Im Standardfall werden Groß- und Kleinschreibung bei der Suche ignoriert; durch die Benutzung von Großbuchstaben im Suchbegriff kann man aber erreichen, dass auch die Groß-/Kleinschreibung der zu findenden Begriffe berücksichtigt wird.

Bzgl. Sonderzeichen wie z.B. Umlauten ist eine sehr beschränkte Art von Ähnlichkeitssuche möglich; bei Eingabe der „einfachen“ Form eines Zeichens werden nicht nur Namen mit exakt diesem Zeichen gefunden, auch Namen mit passenden „erweiterten“ Zeichen werden zurückgeliefert: Eine Eingabe von „Muller“ findet so nicht nur den exakt passenden Namen „Muller“ sondern auch z.B. „Müller“. Durch die Verwendung von HTML–Entities (wie „ü“, „á“, etc.) kann man dieses Verhalten umgehen: Eine Eingabe von „Müller“ findet wirklich nur „Müller“.

Als Ergebnis der Autorensuche erhält man eine Liste der passenden Autoren, jeweils mit einem Verweis auf die entsprechende Autorenseite, bzw. wird direkt auf die Autorenseite weitergeleitet falls sich nur ein Treffer ergibt.

Als Ergebnis der Titelsuche erhält man eine Liste der passenden Publikationen, jeweils mit den zugehörigen, wichtigsten Informationen sowie Verlinkungen auf die entsprechenden Autoren- sowie Konferenz- bzw. Zeitschriften–Seiten.

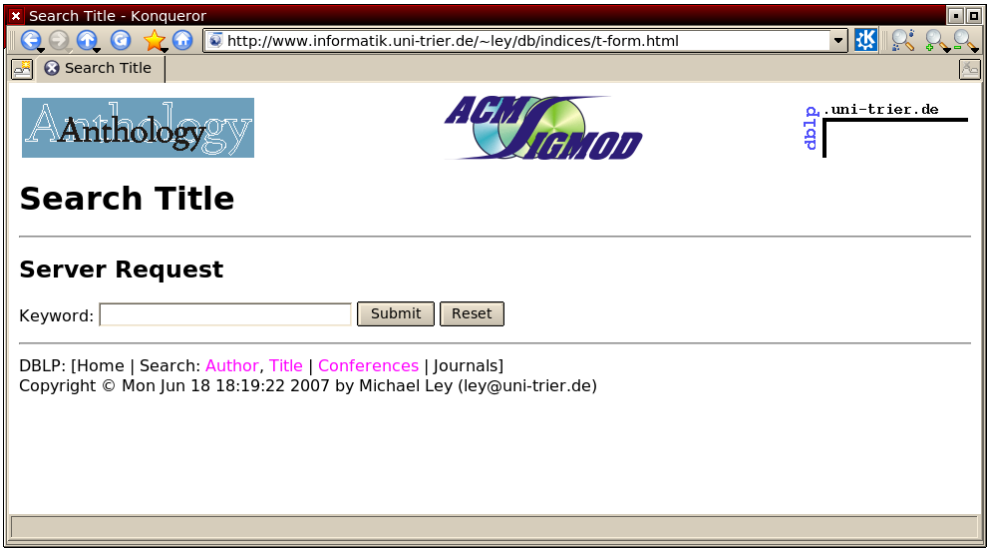

Abbildung 1.2: Titelsuche auf der DBLP–Weboberfläche

1.2.3 Erweiterte Suche mittels Weboberfläche

Als dritte Möglichkeit des Datenzugriffs gibt es schließlich noch die Erweiterte Suche („Advanced Search“). Diese ermöglicht die kombinierte Suche in (fast) allen Datenfeldern der Datensätze.

Neben der Möglichkeit bis zu fünf Autorennamen („Authors“) anzugeben, können dort weiter noch Suchbegriffe zum Titel („Title“), dem (Erscheinungs)jahr („Year“), den Seiten auf denen sich die gesuchte Publikation befindet („Page“), der Konferenz („Conference“) und/oder Zeitschrift („Journal“) anlässlich oder in der die Publikation erschienen ist, die Ausgabe („Volume“) und eine evtl. weitere Kategorisierung/Einteilung („Number“) spezifiziert werden. Außerdem kann durch Eingabe eines Wertes im Feld „Id“ auch nach einer Publikation mit einem spezifischen DBLP–Schlüssel (→ 2.2.2) gesucht werden.

Bei der Suche handelt es sich ebenfalls um eine Substring–Suche. Für die Suche nach Autorennamen wird die Behandlung von Groß-/Kleinschreibung und Sonderzeichen/Umlauten analog wie bei den einfachen Suchen gehandhabt; für die anderen Felder scheint Groß-/Kleinschreibung immer ignoriert zu werden.

Alle angegebenen Suchbegriffe werden automatisch UND–verknüpft, d.h. es werden nur solche Datensätze gefunden, in denen *alle* Suchbegriffe in den entsprechenden Datenfeldern des Datensatzes vorkommen.

Gibt man lediglich in genau einem der „Authors“–Felder einen Suchbegriff an, so erhält man, wie bei der einfachen Autorensuche, eine Liste der passenden Autoren, jeweils mit einem Verweis auf die entsprechende Autorenseite.

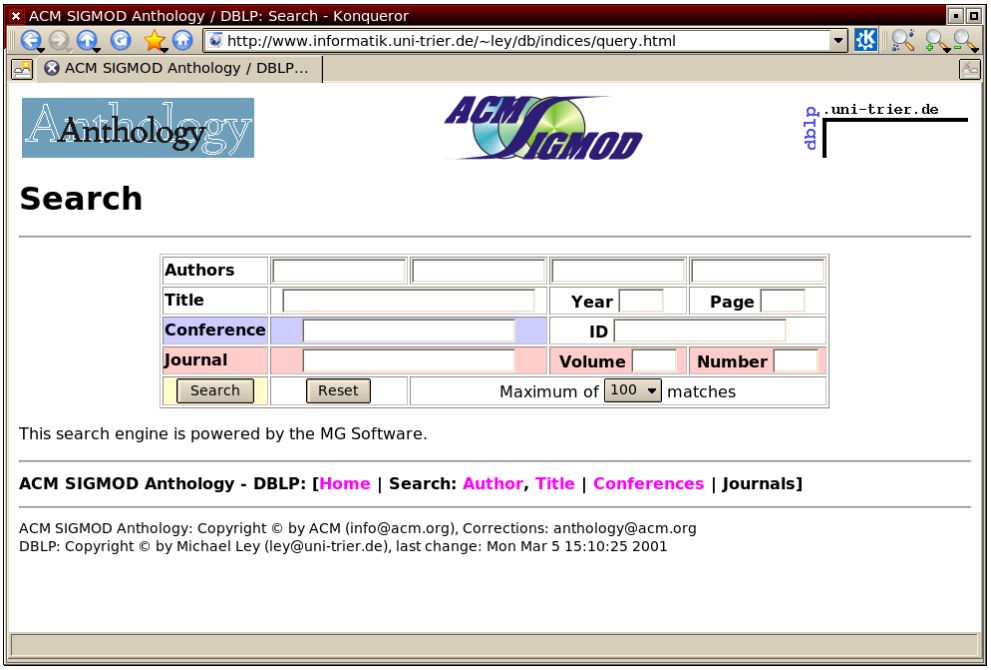

Abbildung 1.3: Erweiterte Suche auf der DBLP–Weboberfläche

In allen anderen Fällen wird nach zu den gegebenen Suchkriterien passenden Publikationen gesucht und man erhält als Ergebnis eine Liste der entsprechenden Publikationen, ebenfalls analog zur Trefferliste der einfachen Titelsuche aufgebaut.

Die Anzahl der anzuzeigenden Ergebnisse kann nach Wunsch auf maximal 200, 100, 50 oder lediglich 10 Stück nach oben hin begrenzt werden.

1.2.4 Browsing mittels DBL Browser

Der DBL Browser ist eine experimentelle Applikation zum Browsen der Datenbestände digitaler bibliographischer Datensammlungen. Er wurde ursprünglich speziell für das DBLP entwickelt und ermöglicht, den Datenbestand offline, auf Basis einer lokal vorhandenen Datensammlung, zu nutzen.

Die Struktur und Darstellung der Daten folgt weitestgehend dem Beispiel der Website; so gibt es z.B. ebenfalls eine Liste der Konferenzen, eine Liste der Zeitschriften, darunter chronologisch geordnete Übersichten zu Veranstaltungen bzw. Ausgaben, Inhaltsverzeichnisse und Autorenseiten. Das Layout dieser Sichten entspricht dem der jeweiligen Webseiten.

Im Gegensatz zur Website stellte der DBL Browser noch drei weitere Indizes

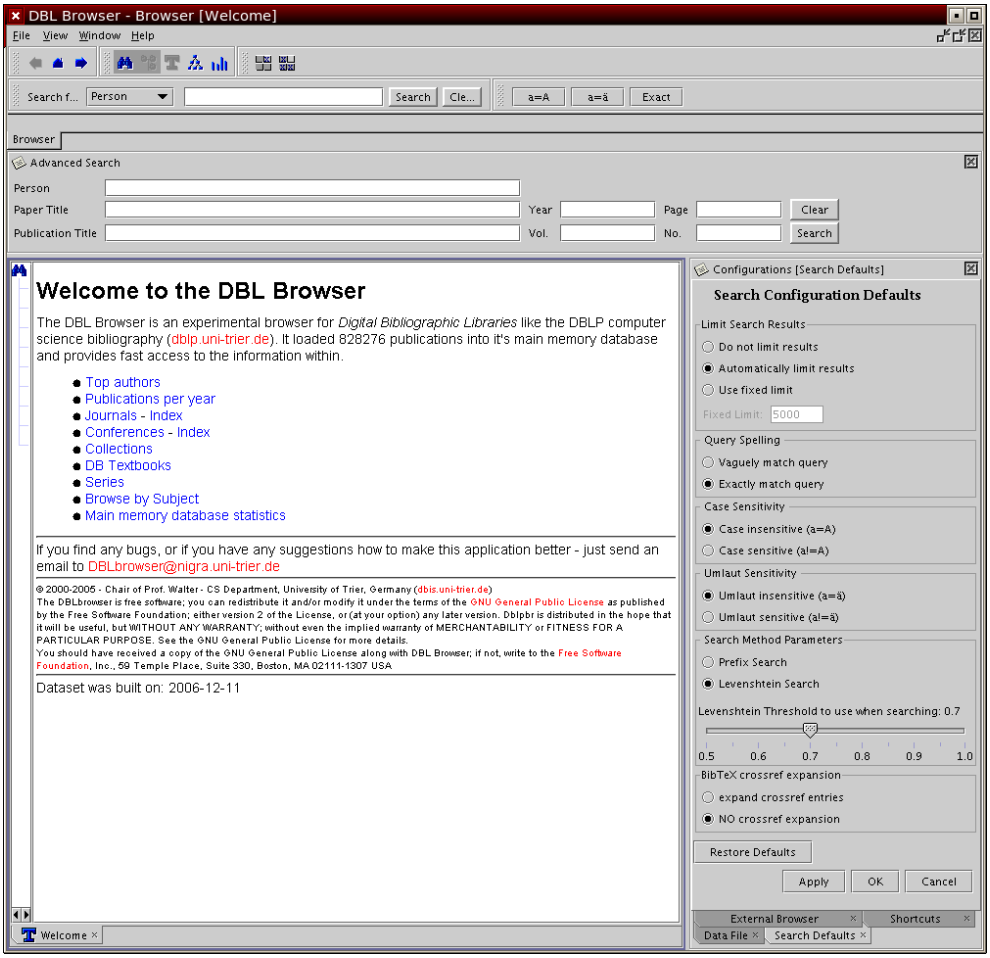

Abbildung 1.4: DBL Browser mit eingeblendeter Erweiterter Suche und Konfigurationsdialog

zur Verfügung: „Collections“, „DB Textbooks“ sowie „Series“. Diese verwalten entsprechend eine Liste von Sammelwerken, Büchern und Buchreihen.

Außerdem enthält der DBL Browser noch einige weitere Statistiken zum DBLP wie z.B. eine Liste der Autoren mit den meisten Veröffentlichungen oder eine Statistik über die Anzahl der Veröffentlichungen aufgeschlüsselt nach Erscheinungsjahr. Weiterhin ist eine graphische Aufbereitung verschiedener Statistiken als Histogramme verfügbar.

1.2.5 Einfache Suche mittels DBL Browser

Der DBL Browser stellt ebenfalls eine einfache Suchmöglichkeit zur Verfügung, die auch in etwa der Autoren- bzw. Titelsuche der Weboberfläche entspricht. Wie diese erlaubt sie (wenn auch durch Anwählen von Buttons statt durch veränderte Eingabe der Suchbegriffe) die Berücksichtigung bzw. Nichtberücksichtigung von Groß-/Kleinschreibung sowie die spezielle Behandlung von Sonderzeichen.

Neben der Möglichkeit nach Autoren und Titeln zu suchen, dient diese einfache Suche außerdem noch dem Auffinden von Konferenzen bzw. Zeitschriften.

Bei der Suche handelt es sich, wie auf der Website auch, um eine einfache Substring–Suche. Als weiteres Feature, welches auf der Website wiederum nicht verfügbar ist, besteht jedoch die Möglichkeit zwischen einer exakten Suche und einer Ähnlichkeitssuche umzuschalten. Hierbei ist allerdings zu beachten, dass als Suchbegriff mindestens vier Zeichen eingegeben werden müssen, ansonsten werden keine Ergebnisse zurückgeliefert.

Der Konfigurationsdialog erlaubt es, die Anzahl der zurückgelieferten Ergebnisse nach den Wünschen des Nutzers zu konfigurieren; allerdings scheinen die dort gemachten Einstellungen keine Wirkung zu haben, es werden offensichtlich immer alle gefundenen Ergebnisse ausgegeben.

1.2.6 Erweiterte Suche mittels DBL Browser

Schlussendlich enthält auch der DBL Browser eine Erweiterte Suche („Advanced Search“). Diese erlaubt es, wie ihr Pendant im Web, nach Titel („Paper Title“), dem (Erscheinungs)jahr („Year“), den Seiten auf denen sich die gesuchte Publikation befindet („Page“), der Konferenz bzw. Zeitschrift („Publication Title“) sowie nach Ausgabe („Vol.“) und Nummer („No.“) zu suchen.

Ein wichtiger Unterschied ist jedoch, dass es hier nicht möglich ist, nach *mehreren* Autoren gleichzeitig zu suchen. Auch die Suche nach Publikationen mit einem bestimmten DBLP–Schlüssel ist nicht möglich.

1.3 Unzulänglichkeiten der bisherigen Suchmöglichkeiten

Wie man sehen kann, existieren offensichtlich bereits eine Reihe von Optionen für die Suche nach Informationen innerhalb des DBLP. Welchen Sinn könnte es also haben, eine weitere Alternative zu entwickeln?

Ein Grund ist dass, wie in den Kurzbeschreibungen oben bereits angeklungen

ist, alle bisherigen Suchmöglichkeiten einigen — teilweise recht restriktiven — Beschränkungen unterliegen, welche die Nützlichkeit dieser Suchen reduzieren. Diese Unzulänglichkeiten werden im Folgenden noch einmal etwas ausführlicher kurz zusammengefasst.

Auf das Browsing wird in diesem Zusammenhang nicht mehr weiter eingegangen, da das Ziel dieser Arbeit die Entwicklung einer *maschinellen* Suche im DBLP–Datenbestand ist. Ungeachtet einer Einbindung der vorhandenen Webseiten in die neue Suchanwendung spielen die Schwächen des „Suchvorgangs“ im dortigen Bereich keine Rolle mehr, da die neue Suche hier vollkommen eigenständig arbeitet.

1.3.1 Autoren- bzw. Titelsuche im Web

- Es handelt sich um eine exakte Substring–Suche. Die gesamte Eingabe wird als eine einzelne Phrase bzw. Zeichenkette aufgefasst und muss (sogar bis auf die genaue Anzahl evtl. eingegebener Leerzeichen zwischen Begriffen!) genau in dieser Schreibweise im Suchfeld (Autor, bzw. Titel) vorkommen.

- Andererseits wird Groß-/Kleinschreibung bei der Suche im Normalfall ignoriert; es ist zwar möglich, durch Benutzung von Großbuchstaben die Berücksichtigung der Schreibweise zu erzwingen, allerdings ist diese Herangehensweise etwas unintuitiv. Des weiteren kann auf diese Weise nicht nach Begriffen vollständig in Kleinbuchstaben gesucht werden.

- Dass bei Eingabe von „einfachen“ Zeichen auch verwandte Sonderzeichen bzw. Umlauten gefunden werden, ist angesichts der ansonsten sehr exakten Suche ebenfalls weniger intuitiv. Das Erzwingen einer exakten Suche durch Eingabe in Entity–Form ist recht umständlich.

- Die Anzahl der zurückgelieferten Ergebnisse ist fest auf max. 100 Resultate beschränkt.

- Da es sich um eine rein sequentielle Suche im Datenbestand handelt[7] ist die Performance, selbst bei nur einem Nutzer, alleine schon aufgrund der Größe der Datenbasis nicht wirklich befriedigend.[8]

- Die Reihenfolge der zurückgelieferten Ergebnisse wird (bei der Titelsuche) nur dadurch bestimmt, in welcher Reihenfolge die Datensätze in der Datei

[7] Wobei allerdings eine speziell generierte Liste zum Einsatz kommt; vgl. [DFAQ]

[8] Eigene, informelle Tests ergaben eine durchschnittliche Suchdauer von ca. 2 Sekunden bei einfachen Abfragen nach einem Autorennamen, in Einzelfällen stieg die Dauer aber sogar bis auf ca. 15 Sekunden an.

stehen. Ein nach sinnvollen Kriterien gewichtetes Ranking findet augenscheinlich nicht statt. Die Ergebnisse der Autorensuche werden alphabetisch sortiert.

- Die Suche ist mit Autor(en) und Publikationstitel auf nur zwei — wenn auch wohl die wichtigsten — der vorhandenen Datenfelder beschränkt.

1.3.2 Erweiterte Suche im Web

- Aufgrund der teilweise gleichen Funktionsweise der Erweiterten Suche wie bei den einfachen Suchen gelten die Aussagen zur exakten Suche, zur Berücksichtigung von Groß-/Kleinschreibung und zur Behandlung von Sonderzeichen/Umlauten (s.o.) entsprechend.

- Bei der Eingabe mehrere Suchbegriffe werden diese automatisch mittels UND verknüpft. Eine Suche nach Publikationen, in denen nur mindestens einer der Suchterme vorkommt ist daher nicht möglich.[9]

- Auch die erweiterte Suche erlaubt es noch nicht, alle vorhandenen Datenfelder zu durchsuchen. Zwar enthalten die nicht durchsuchbaren Felder nur einen vergleichsweise geringen Anteil der DBLP–Daten (→ 2.3.2), aber es ist zumindest nicht auszuschließen, dass in bestimmten Situationen auch die Suche nach diesen Informationen relevant sein könnte.

- Das Programm, mittels welchem die Erweiterte Suche realisiert ist, setzt auf einer spezifischen, mittlerweile veralteten Version einer Software–Bibliothek auf, die in neueren Betriebssystem–Versionen nicht mehr verfügbar ist. Aus diesem Grund ist fraglich, wie lange diese Software noch eingesetzt werden kann.[10]

1.3.3 Suchen (einfach und erweitert) im DBL Browser

- Es ist zwar möglich, im Konfigurationsdialog die gewünschte Anzahl der zurückzuliefernden Suchergebnisse einzustellen, anscheinend wird dieser Parameter aber bisher nicht berücksichtigt und es werden immer alle gefundenen Resultate angezeigt.

[9] Der Nutzer könnte dieses Verhalten im Prinzip zwar durch mehrere einzelne Suchen mit jeweils nur einem der Suchterme simulieren, allerdings müssten die erhaltenen Ergebnisse dann umständlich manuell zusammengeführt werden.

[10] Aussage Michael Ley

- Um das Programm nutzen zu können, muss der gesamte DBLP–Datenbestand heruntergeladen werden, was in Anbetracht der großen Menge der Daten u.U. ziemlich aufwändig ist.
- Die lokal zur Verfügung stehenden Daten sind u.U. nicht auf dem neuesten Stand. Um immer mit den aktuellsten Daten arbeiten zu können, müssten diese im Prinzip vor jeder Benutzung komplett neu heruntergeladen werden.
- Um den DBL Browser benutzen zu können muss nicht nur das Programm an sich heruntergeladen und installiert werden, sondern es ist auch eine entsprechend aktuelle Version der Java–Runtime–Umgebung nötig, die ggf. zusätzlich heruntergeladen und installiert werden muss.
- Die Startzeit des DBL Browser ist verhältnismäßig hoch[11], da die Daten erst komplett eingelesen und analysiert werden müssen, bevor sie zur Benutzung zur Verfügung stehen. Dies macht den DBL Browser für den spontanen Einsatz eher ungeeignet.

1.3.4 Erweiterte Suche im DBL Browser

- Auch bei der Erweiterten Suche im DBL Browser werden, wie im Web, mehrere Suchbegriffe immer automatisch mit UND verknüpft.
- Auch hier ist es, wie im Web, nicht möglich alle vorhandenen Datenfelder zu durchsuchen.

1.4 Sinnvolle oder wünschenswerte neue Features für die Suche

Neben den oben genannten Beschränkungen existieren eine Reihe von denkbaren neuen Features und Erweiterungsmöglichkeiten, welche die Arbeit mit dem DBLP erleichtern könnten.

Es handelt sich hierbei vor allem um eine Reihe weiterer nützlicher Typen von Suchen, die insb. in speziellen Fällen und aufgrund der spezifischen im DBLP vorhandenen Daten und deren Eigenschaften von großem Nutzen sein könnten:

[11] Ca. 40 Sekunden auf dem Testsystem.

1.4.1 Ähnlichkeitssuche

Ein großes Problem vieler Datensammlungen, so auch des DBLP, ist dass Namen, insb. auch Personennamen, oftmals in vielen unterschiedlichen Schreibweisen eingegeben wurden. Bei Eingabe eines Suchbegriffes würden in einem solchen Fall normalerweise nur diejenigen Datensätze gefunden, bei denen die Daten exakt mit dem eingegebenen Suchbegriff übereinstimmen.

Im Unterschied dazu werden bei einer Ähnlichkeitssuche („fuzzy search“) hingegen auch Begriffe gefunden, die dem Suchbegriff lediglich ähneln; eine exakte Übereinstimmung der Schreibweise ist nicht nötig.

Für die DBLP–Suche könnte sich die Möglichkeit zur Ähnlichkeitssuche insb. im Hinblick auf die Autoren(namen) als nützlich erweisen, da gerade hier — trotz großer Bemühungen zur Vereinheitlichung — oftmals noch mehrere unterschiedliche Schreibweisen zu finden sind.[12]

Spezielle Formen bzw. Erweiterungen der Ähnlichkeitssuche sind Suche mit Rechtschreibkorrektur und Suche mittels Phonetischer Ähnlichkeit.

Bei der Suche mit Rechtschreibkorrektur werden ähnliche Schreibweisen der Suchterme evtl. nur unter bestimmten Umständen in Betracht gezogen, z.B. wenn nur sehr wenige Resultate gefunden werden konnten oder der Suchterm in der eingegebenen Schreibweise nicht im Suchindex vorhanden ist.[13]

Bei der Phonetischen Suche werden spezielle Algorithmen[14] verwendet, die statt Ähnlichkeiten der Schreibweise eher Ähnlichkeiten des Klangs bzw. der Aussprache berücksichtigen. Insb. wiederum bei der Suche nach Namen, bei denen es oft viele gleich klingende Varianten gibt und die exakte Schreibweise deshalb oft nicht bekannt ist, könnte diese Suchvariante gute Dienste leisten.

1.4.2 Synonymsuche

Oftmals ist die Suche nach Informationen zu einem bestimmten Thema einfach aus dem Grund nicht erfolgreich, weil verschiedene gleichwertige Bezeichnungen für eine Sache existieren und der Nutzer eine andere Terminologie und bei der Suche eine andere als die in der Datenbank vorhandene Bezeichnung verwendet.

Um diesem Problem entgegenzuwirken bietet es sich an, bei der Suche automatisch auch nach Synonymen der eingegebenen Suchbegriffe suchen zu lassen.[15]

[12] vgl. z.B. [LR06], S. 44ff.

[13] vgl. z.B. [MRS07], S. 43ff.

[14] Bekannt sind Soundex, siehe z.B. http://en.wikipedia.org/wiki/Soundex oder Metaphone, siehe z.B. http://en.wikipedia.org/wiki/Metaphone

[15] vgl. z.B. [R06nw], Abschnitt „6. Implement synonyms“

Im Rahmen der DBLP–Suche ist diese Möglichkeit zwar nur im Rahmen der Suche nach Publikationstiteln sinnvoll, könnte dort aber das Suchergebnis in bestimmten Situationen verbessern. Bei der Suche nicht nach einer bestimmten Publikation oder einem bestimmten Autor, sondern allgemein nach Informationen zu einem bestimmten Themengebiet beispielsweise lässt sich auf diese Weise wahrscheinlich die Anzahl der insgesamt gefundenen Publikationen erhöhen.[16]

Bei allen anderen Datenfelder des DBLP handelt es sich um Namen bzw. andere Bezeichner, die keine direkte semantische Bedeutung in sich tragen und bei denen aus diesem Grund die Anwendung von Synonymen nicht möglich bzw. nicht sinnvoll ist. Bei diesen Feldern macht stattdessen eher die Anwendung einer Ähnlichkeitssuche zur Erfassung eventueller unterschiedlicher Schreibweisen Sinn.

1.4.3 Bereichs–Suche

Bei Feldern, die einen kontinuierlichen Wertebereich aufweisen — insb. Jahres- und andere Zahlen — kann es evtl. von Interesse sein, Datensätze zu finden, bei denen der entsprechende Wert innerhalb eines bestimmten Intervalls liegt.

Bei der DBLP–Suche ist z.B. durchaus denkbar, dass sich ein Nutzer für Publikationen aus einem ganz bestimmten Zeitraum interessiert; in einem solchen Fall wäre es schön, wenn man das Intervall, in dem das Erscheinungsjahr der Publikationen liegen soll, explizit spezifizieren könnte. Auch für die Felder „number“ und „volume“ könnte eine solche Suche Sinn machen, wenn man nur ungefähr weiß, in welcher Ausgabe z.B. einer Zeitschrift sich eine gesuchte Publikationen befindet. In den Feldern „month“ und „chapter“ auf diese Weise zu suchen wäre ebenfalls möglich, allerdings sind hier bislang nur für sehr wenige Datensätze im DBLP Daten verfügbar (vgl. Abschnitt 2.3.2).

1.4.4 Proximity–Suche

Bei Suchbegriffen, welche aus zwei oder mehr Wörtern bestehen, kann es durchaus vorkommen, dass es bzgl. der Reihenfolge und Anordnung der Einzelwörter mehrere unterschiedliche gültige Varianten gibt. Auch können u.U. (aus grammatikalischen Gründen) je nach Schreibweise noch andere Wörter zwischen den gesuchten Teilbegriffen stehen.

Bei Personennamen ergibt sich ebenfalls ein entsprechendes Problem da z.B. die Stellung bzw. Reihenfolge von Ruf- und Familienname in verschiedenen Kul-

[16] Was allerdings ggf. auch von Nachteil sein kann, falls dadurch viele irrelevante Dokumente zurückgeliefert werden; vgl. z.B. [R79], Kapitel 7, [MRS07], S. 118ff oder auch [WPrp].

turen verschieden gehandhabt wird oder zusätzliche Vornamen aufgeführt oder weggelassen werden und so unterschiedliche Schreibweisen des an sich gleichen Namens entstehen können.

Bei der Proximity–Suche können diese Probleme dadurch umgangen werden, dass keine zusammenhängende Phrase angegeben werden muss, bei der Reihenfolge der Wörter und auch die Wörter an sich exakt festgelegt sind. Stattdessen spezifiziert man die Einzelwörter sowie die Entfernung (in Wörtern, seltener evtl. auch Zeichen) innerhalb derer diese Wörter vorkommen müssen.[17]

für die DBLP–Suche könnte sich dieses Verfahren insb. Für die Suche nach Autoren(namen) und bei der Suche nach Publikationen (Titeln) zu bestimmten Themen als nützlich erweisen.

1.4.5 Term Boost

Die Einordnung der gefundenen Suchergebnisse nach ihrer (wahrscheinlichen) Relevanz („ranking“) wird normalerweise implizit von der Suchmaschine nach „sinnigen“ Kriterien vorgenommen. Bei der Ähnlichkeitssuche z.B. werden im Normalfall diejenigen Ergebnisse an erster Stelle platziert, in denen der gefundene Begriff in exakt gleicher oder am wenigsten abweichenden Schreibweise vorkommt.

U.U. kann es aber auch wünschenswert sein, bei der Suche nach mehreren Begriffen diese unterschiedlich zu gewichten und so selbst das Ranking zu beeinflussen.[18] Bei der Synonymsuche dagegen könnten automatisch hinzugefügte Terme geringer gewichtet werden, als die Original–Suchterme.[19]

Bei der DBLP–Suche wäre es z.B. denkbar, dass der Nutzer zwar an den Publikationen mehrere Autoren interessiert ist, einem dieser Autoren jedoch besondere Bedeutung zuweist und deshalb die Publikationen dieses Autors höher bewertet haben möchte. Auch bei der Suche nach Publikationstiteln ist es durchaus wahrscheinlich, dass zu einem bestimmten, vom Nutzer gewünschten Themengebiet, mehrere Suchbegriffe passend sind, einer davon jedoch die Thematik in besonderer Weise wiederspiegelt und deswegen höher bewertet werden soll.

1.4.6 Suche mit Platzhaltern oder Regulären Ausdrucken

Die Suche mit Platzhaltern bzw. — quasi als fortgeschrittene Version davon — die Suche mittels regulärer Ausdrücke erlaubt dem Nutzer auf syntaktischer Ebene

[17] vgl. z.B. [WPps].
[18] vgl. [ASFLqs], Abschnitt „Boosting a Term“.
[19] vgl. [MRS07], S. 149.

eine weitergehende Spezifikation der zu findenden Begriffe.[20]

Bei der Suche mittels Platzhalten bezeichnen spezielle Zeichen (die Platzhalter oder „Wildcards“) die Stellen im Suchbegriff, deren Inhalt variabel bleiben soll. Meist werden das Asterisk „*“ verwendet, wenn man festlegen will, dass an der betreffenden Stelle ein oder mehrere Zeichen variabel sein sollen und das Fragezeichen „?“ um exakt ein beliebiges Zeichen zu erlauben. Reguläre Ausdrücke erlauben, je nach Mächtigkeit der Implementierung, noch wesentlich weitergehende Angaben.[21]

Im Prinzip könnte man diese Art der Suche als eine Art „manueller Ähnlichkeitssuche“ bezeichnen; statt sich auf feste, automatische Verfahren zur Bestimmung der Ähnlichkeit der Zeichenketten zu stützen kann bzw. muss der Nutzer selber definieren wo und welche Unterschiede in der Schreibweise der Suchbegriffe auftreten dürfen.

Dies ermöglicht einerseits natürlich eine größere Kontrolle der Suche; andererseits erfordert die Definition korrekter Suchmuster — insb. bei regulären Ausdrücken — u.U. einige Fachkenntnisse und wird sich in den meisten Fällen wohl für den „normalen Nutzer“ als zu schwierig und aufwändig erweisen.

1.5 Konkrete Zielsetzung dieser Arbeit

Ziel dieser Diplomarbeit ist es, eine neue Suchmaschine für das DBLP zu entwickeln. Diese soll die von bisherigen Suchen angebotenen Funktionen einschließen und dabei die dort bestehenden Beschränkungen reduzieren oder am Besten ganz vermeiden. Darüber hinaus soll sie außerdem neue, bislang nicht verfügbare Features bereitstellen um die Arbeit mit dem DBLP weiter zu erleichtern und das Potential der Datenbank in breiterem Umfang nutzbar zu machen.

Im Einzelnen stellen sich die Anforderungen und Erweiterungswünsche wie folgt dar (grob sortiert nach Priorität, angefangen mit den wichtigsten Punkten):

- *Alle Daten durchsuchbar:* Es sollen alle oder, falls das nicht möglich oder nur mit zu hohem Aufwand zu realisieren wäre, zumindest alle wichtigen Datenfelder durchsuchbar sein. Die Suche soll den gesamten verfügbaren Datenbestand, d.h. alle erfassten Publikationen, einschließen.

- *Aktualität der Daten:* Die Suche soll auf einem so aktuell wie möglich gehaltenen Datenbestand, am Besten tagesaktuell oder besser, aufsetzen. Dies

[20] vgl. z.B. [MRS07], S. 39ff.

[21] Ein bekanntes Format sind die sog. Perl Regular Expressions, die aus der gleichnamigen Skriptsprache stammen, mittlerweile jedoch auch in anderen Bereichen weit verbreitet sind; siehe [Cpre].

erfordert u.A., dass die Erstellung des Suchindex (→ 4.1.2) in annehmbarer Zeit möglich ist.

- *Gute Performance:* Die Performance der Suche soll, auch bei Zugriff durch mehrere Nutzer gleichzeitig, ausreichend schnell sein, so dass keine spürbaren Verzögerungen entstehen.

- *Ranking der Suchergebnisse:* Die Suchergebnisse sollen nach einem sinnvollen Verfahren bewertet und sortiert werden. Die Präsentation der Ergebnisse soll ebenfalls diesem Ranking gemäß erfolgen und dem Nutzer eine Rückmeldung bzgl. der Qualität der einzelnen Treffer geben.

- *Schnell und einfach verfügbar:* Die Suche soll ohne großen Aufwand oder Vorbereitung verfügbar sein; die Installation von Zusatzprogrammen oder das Herunterladen von großen Datenmengen soll nicht notwendig sein. Am einfachsten lässt sich dieses Ziel wohl durch Erstellen einer Webanwendung erreichen, welche dann von überall her im Internet verfügbar ist.

- *Neue Suchtypen:* Sofern es einzurichten ist, sollen weitere Suchtypen wie Ähnlichkeitssuche, Wildcard–Suche, etc. zusätzlich angeboten werden. Insb. eine Ähnlichkeits- oder Phonetische Suche (auch) für die Autorennamen und eine Proximity–Suche ebenfalls (auch) für Autorennamen und auch Publikationstitel scheint hier vielversprechend. Synonymsuche (insb.) für Publikationstitel und Bereichssuche (auch) für Erscheinungsjahr könnten ebenfalls hilfreich sein.

- *Volle Unterstützung für Boolean–Queries bzw. Kombination von Query–Typen:* Die Angabe mehrerer Suchbegriffe mit Bestimmung, welche Felder durchsucht werden sollen soll möglich sein; ebenso Verknüpfung dieser Terme nach Wunsch mit UND oder ODER sowie Kombinationen davon; außerdem Gruppierung mittels Klammerung. Wenn möglich sollen die bereitgestellten Suchtypen gleichzeitig nebeneinander benutzt und innerhalb eines Queries kombiniert werden können.

- *Kompatibilität der Anwendung:* Die Anwendung soll, soweit machbar, unter vielen Betriebssystemen und Systemversionen zu bauen und verwendbar sein. Soweit erforderlich sollte sie auf relativ bekannten Bibliotheken und Tools aufsetzen, so dass auch die zukünftige Entwicklung und Lauffähigkeit gewährleistet ist.

- *Konfigurierbare Ergebnisliste:* Die zu präsentierende Anzahl und evtl. auch die Darstellung der Suchergebnisse sollen vom Nutzer seinen Wünschen

nach angepasst bzw. ausgewählt werden können. Evtl. Umstellung auf seitenweise Anzeige der Suchergebnisse.[22]

- *Bessere Steuerung:* Berücksichtigung von Groß-/Kleinschreibung bei der Suche oder die Behandlung von Sonderzeichen sollen (wenn diese Features überhaupt interessant sind) auf einfache und intuitive Weise gesteuert werden können.

- *Weitere Features:* Weitere Features wie Term Boost, Wildcard- bzw. Suche mit regulären Ausdrücken, etc.

[22] Wie u.A. aus eigener Erfahrung bekannt, wird die Möglichkeit zur Veränderung der Anzahl der gelieferten Ergebnisse eher selten genutzt; vgl. auch z.B. [R05wti], Abschnitt „What not to include“. Andererseits ist eine Begrenzung der zurückgelieferten Resultate erforderlich, um bei allzu allgemeinen Abfragen, die sehr viele Treffer liefern würden, die Ressourcen des Systems nicht zu sehr zu belasten. Eine seitenweise Anzeige scheint hier ein guter Kompromiss.

Kapitel 2

Analyse der DBLP–Daten

Da die genaue Art und Struktur der zu durchsuchenden Daten und das Format, in dem diese vorliegen, für die weitere Arbeit eine grundlegende Rolle spielen, steht eine eingehende Analyse dieser Aspekte am Anfang der Arbeit.

Welche Werkzeuge bei der Implementierung der neuen Suche zum Einsatz kommen können oder müssen, welche Änderungen oder Konvertierungen nötig und welche der vorher gesteckten Ziele mit wieviel Aufwand erreichbar sind hängt teilweise entscheidend von diesen Gegebenheiten ab.

In diesem Kapitel werden daher die verschiedenen physikalischen bzw. Dateiformate, in denen der DBLP–Datenbestand verwaltet wird, vorgestellt und ihre Eignung als Grundlage für die neue Suchanwendung bewertet. Das Format der XML–Datei `dblp.xml` und die Punkte, die bei der Auswertung eine Rolle spielen, werden genauer untersucht. Schließlich wird auf die Daten selbst eingegangen und beschrieben, welche Bereiche der Suche hiervon beeinflusst werden.

2.1 Verfügbare physikalische Formate

Der Datenbestand des DBLP liegt z.Zt. in drei verschiedenen Formaten vor:[1]

- Die Ausgangsdaten befinden sich in einer Sammlung von XML–Dateien; für jede im DBLP erfasste Publikation existiert eine Datei, welche die vorhandenen Informationen zur entsprechenden Publikation speichert.
- Aus diesen einzelnen XML–Dateien wird einmal täglich eine Große XML–

[1] vgl. [DFAQ] oder [V06], S. 5ff.

Datei `dblp.xml`[2] generiert, welche die Daten zu allen Publikationen zusammenfasst.

- Als Drittes sind die Daten natürlich auch in den HTML–Seiten der DBLP–Weboberfläche enthalten.

Im Prinzip könnten alle drei Varianten als Grundlage für die Suche bzw. Erstellung des Suchindex dienen. Es zeigt sich jedoch schnell, dass die einzelnen Formate hierfür teilweise erhebliche Vor- bzw. Nachteile mit sich bringen.

2.1.1 HTML–Seiten

Theoretisch wäre es natürlich möglich, auf den HTML–Seiten der DBLP–Website aufzusetzen und diese als Datengrundlage für die Suche zu benutzen. Allerdings finden sich hier sofort eine ganze Reihe von Nachteilen, die diese Alternative als nicht sinnvoll erscheinen lassen:

- Es handelt sich um einzelne Dateien bzw. Seiten, die jeweils einzeln geladen und behandelt werden müssen. Auch wenn das an sich kein großes Hindernis darstellt bringt es doch einen zumindest geringfügig höheren Aufwand mit sich.

- Im Zusammenhang damit stellt sich allerdings ein gravierenderes Problem: Die Struktur der Website bzw. der Dateiensammlung an sich ist verhältnismäßig kompliziert und es gibt keinen Gesamtindex, der alle Dateien in einer einfachen Art und Weise referenziert.

  Als sinnvollster Einstiegspunkt hierfür würde sich wahrscheinlich noch der Autorenindex anbieten, wobei hier zu beachten wäre, dass nicht für alle Publikationen ein Autor gespeichert ist[3] und aus diesem Grund auch nicht alle Daten darüber erreichbar wären.

- Dieses Problem wiederholt sich im kleinen Maßstab, da auch die Struktur der einzelnen Seiten an sich nicht immer einheitlich ist und teilweise manuell eingefügte Zusatzinformationen vorhanden sind.

- Am schwersten wiegt aber die Tatsache, dass es sich bei HTML um eine Seitenbeschreibungssprache handelt und damit — soweit es die DBLP-Daten betrifft — die semantische Struktur der Informationen weitestgehend nicht mehr vorhanden ist.

[2] Öffentlich verfügbar unter `http://dblp.uni-trier.de/xml/dblp.xml`, Größe ungepackt mittlerweile (Juli 2007) über 400 MiB.

[3] siehe Tabelle 2.2

Offensichtlich ist es, soweit es überhaupt möglich wäre, mit erheblichem Aufwand verbunden und daher kaum praktikabel, die Daten aus den HTML–Seiten zu extrahieren.

Allenfalls im Rahmen einer unstrukturierten Volltext–Suche könnte es evtl. Sinn machen, die HTML–Dateien als Datenbasis heranzuziehen. Dies würde es ermöglichen, quasi im Sinne einer allgemeinen Website–Suche, nicht nur die eigentlichen DBLP–Publikationen–Daten, sondern auch noch die teilweise manuell auf den Webseiten ergänzten Zusatzinformationen zu finden.

Da diese Zusatzinformationen jedoch, sowohl von ihrer Menge als auch von ihrer Bedeutung her, nur einen sehr geringen Teil des DBLP ausmachen, ist diese Option bestenfalls als Zusatzangebot mit niedriger Priorität anzusehen und wird im Rahmen dieser Arbeit nicht weiter verfolgt.

2.1.2 Publikationen–XML–Dateien

Direkt auf den einzelnen Publikationen–Dateien aufzusetzen erscheint dagegen als eine wesentlich sinnvollere Alternative. Auf der Basis dieser Dateien zu arbeiten vermeidet die Nachteile der HTML–Dateien und bietet folgende Vorteile:

- *Aktuell:* Da es sich um die Ursprungsdaten handelt, in denen alle Änderungen und Ergänzungen direkt eingepflegt werden, liegt hier jederzeit der aktuellste Datenbestand vor.
- *Komplett:* Aus demselben Grund umfassen die Dateien den gesamten zu indexierenden Datenbestand des DBLP.
- *Strukturiert:* Die Daten liegen hier gut strukturiert (XML) in einem für alle Publikationen einheitlichen Format vor.

Die Einzeldateien der Publikationen stellen damit bereits eine ansprechende Möglichkeit dar, um als Grundlage zur Erstellung des Suchindex zu dienen.

Als kleiner Nachteil dieses Formates ist allerdings zu sehen, dass diese Dateien normalerweise nicht öffentlich zur Verfügung stehen. In Bezug auf die hier entwickelte „interne“ DBLP–Suche fällt dieser Punkt zwar nicht so sehr ins Gewicht, da anzunehmen ist, dass sowohl der Datenbestand als auch die Suche vom gleichen Personenkreis betreut wird und die Daten für die Administratoren somit jederzeit zugreifbar sind. Allein schon beim Schreiben dieser Diplomarbeit hat es sich jedoch schon als einfacher erwiesen, auf die leichter zugängliche Datei `dblp.xml` zurückzugreifen.

Außerdem handelt es sich hier, wie schon bei den HTML–Seiten, um einzelne Dateien, die ebenso einzeln geladen und behandelt werden müssten, was auch hier einen zumindest geringfügig höheren Aufwand zur Folge hätte.

2.1.3 Datei `dblp.xml`

Als geeignetste Datengrundlage für die Suche kristallisiert sich daher schließlich die große XML–Datei `dblp.xml` heraus. Diese erfüllt alle erforderlichen Punkte:

- *Aktuell:* Da die Datei täglich erstellt wird, enthält sie immer die neuesten, tagesaktuellen Daten.
- *Komplett:* Die Datei umfasst, wie die Publikationen–Dateien, ebenfalls den gesamten zu indexierenden Datenbestand des DBLP, allerdings in kompakterer Form.
- *Strukturiert:* Die Daten liegen auch hier gut strukturiert (XML) in einem für alle Publikationen einheitlichen Format vor.
- *Verfügbar:* Die Datei ist schnell und einfach öffentlich verfügbar.

Aufgrund dieser positiven Bewertung setzt die neue Suchanwendung auf der Datei `dblp.xml` als Datenbasis auf.

2.2 Logisches Format der Datei `dblp.xml`

2.2.1 XML–Struktur und Elementhierarchie

Innerhalb der Datei `dblp.xml` sind die DBLP–Daten nach semantischen Gesichtspunkten strukturiert abgelegt. Die Struktur dieser Datei ist verhältnismäßig einfach gehalten:

Das Grundelement der DBLP–Daten sind die einzelnen Publikationen; für jede erfasste Publikation existiert genau ein zugehöriger Datensatz.

Ein Datensatz besteht aus einem umschließenden XML–Element welches, über den Elementnamen, den Typ der Publikation definiert. Darin enthalten sind Unterelemente, im Folgenden als „Datenfelder“ bezeichnet, in variabler Anzahl und von variablem Typ, welche die spezifischen Informationen zu dieser Publikation verwalten.

Die einzelnen Publikationen–Datensätze sind, ohne bestimmte Reihenfolge, sequentiell hintereinander in der Datei abgelegt.

Das Root-Element „dblp“ umschließt alle Datensätze und garantiert auf diese Weise die Wohlgeformtheit der XML–Datei.

Listing 2.1: Ausschnitt (Anfang und Ende) der Datei `dblp.xml`, Stand 23. Feb. 2007. Aus Gründen der Übersichtlichkeit wurden händig einige Abstände, Umbrüche und Einrückungen eingefügt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>

<incollection mdate="2002-01-03"
              key="books/acm/kim95/AnnevelinkACFHK95">
   <author>Jurgen Annevelink</author>
   <author>Rafiul Ahad</author>
   <author>Amelia Carlson</author>
   <author>Daniel H. Fishman</author>
   <author>Michael L. Heytens</author>
   <author>William Kent</author>
   <title>Object SQL - A Language for the Design and
          Implementation of Object Databases.</title>
   <pages>42-68</pages>
   <year>1995</year>
   <booktitle>Modern Database Systems</booktitle>
   <url>db/books/collections/
        kim95.html#AnnevelinkACFHK95</url>
</incollection>

<incollection mdate="2002-01-03"
              key="books/acm/kim95/Blakeley95">
   <author>Jos&eacute; A. Blakeley</author>
   <title>OQL[C++]: Extending C++ with an Object Query
          Capability.</title>
   <pages>69-88</pages>
   <booktitle>Modern Database Systems</booktitle>
   <url>db/books/collections/kim95.html#Blakeley95</url>
   <year>1995</year>
</incollection>

<!-- ... (weitere Datensaetze) ... -->

<www mdate="2002-01-03"
     key="www/org/oasis-open-cover-xml">
   <author>Robin Cover</author>
```

```
    <title>The SGML/XML Web Page</title>
    <url>http://www.oasis-open.org/cover/xml.html</url>
</www>

<www mdate="2002-01-03" key="www/org/tpc">
    <title>Transaction Processing Performance
           Council</title>
    <url>http://www.tpc.org</url>
</www>

</dblp>
```

Insgesamt ist die Verschachtelungshierarchie der Datensätze max. drei Ebenen (Elemente) tief. Allerdings ist zu beachten, dass in einigen Fällen auch noch innerhalb des „title“–Datenfeld–Elements weitere Unterelemente wie `<i>`, `<sub>`, `<sup>`, `<tt>` oder `<ref>` vorkommen können.[4] Es handelt sich hierbei um Tags, mit Hilfe derer bestimmte Textformatierungen definiert werden. Diese beeinflussen jedoch nicht die Struktur der Daten an sich und können für die Suche deshalb ignoriert werden.

2.2.2 XML–Attribute

Die Elemente der Publikationen verfügen im Normalfall über zwei Attribute:

- `key`: Dieser eindeutige Schlüssel wird für jede Publikation bei der Aufnahme in das DBLP vergeben. Die genaue Zusammensetzung ist je nach Publikationstyp verschieden: so enthalten die Schlüssel für Bücher u.A. das Verlagskürzel, Autorennamen und Jahreszahl; bei Zeitschriftenartikeln finden sich u.A. das Kürzel der Zeitschrift sowie ebenfalls Autorennamen und Jahreszahl; bei WWW–Links wird der Schlüssel aus Bestandteilen der URL gebildet.

  Der Schlüssel kann sowohl innerhalb des DBLP als auch in externen Texten oder Anwendungen zur eindeutigen Bezugnahme auf einen DBLP–Datensatzes verwendet werden. So ist der Schlüssel u.A. Bestandteil der URL der BibTEX–HTML–Seiten (→ 1.2.1) der einzelnen Publikationen.

- `mdate` („Modification DATE“): Das hier enthaltene Datum gibt den Zeitpunkt der letzten Änderung des entsprechenden Datensatzes an.

  Im Kontext dieser Arbeit könnte es unter Umständen dazu verwendet werden, die Suche nach aktuell geänderten oder neu eingefügten Publikationen

[4] vgl. zugehörige Datei `dblp.dtd`

zu erlauben. Außerdem besteht evtl. die Möglichkeit, diese Information zur inkrementellen Aktualisierung des Suchindex heranzuziehen. Hierbei ist allerdings zu beachten, dass es sich laut DTD um ein optionales Attribut handelt, welches also keineswegs bei allen Datensätzen vorhanden sein muss.

Publikationen vom Typ „article“ können darüber hinaus außerdem noch zwei weitere (allerdings optionale) Attribute „reviewid“ und „rating“ enthalten. Artikel–Einträge mit diesen Attributen bezeichnen Besprechungen anderer Artikel bzw. Publikationen. Dabei verweist „reviewid“ über den DBLP–Schlüssel und einen Zusatz auf die besprochene Publikation; „rating“ enthält eine Bewertungsklassifizierung.

Die Elemente der Datenfelder enthalten teilweise ebenfalls noch weitere XML–Attribute:

- Elemente vom Typ „publisher“ enthalten optional ein Attribut „href“ in welchem entweder direkt die URL der Homepage des Verlages oder die URL einer Infoseite dazu innerhalb der DBLP–Website eingetragen ist.
- Elemente vom Typ „series“ enthalten ebenfalls optional ein Attribut „href“ mit der gleichen Bedeutung.
- Elemente vom Typ „cite“ enthalten optional ein Attribut „label“.

Das in der DTD aufgeführte Datenfeld–Element „layout“ (mit optionalem Attribut „logo“) wird z.Zt. noch nicht benutzt.

2.2.3 Bewertung

Insgesamt liegen die Daten hier in einer übersichtlichen und einfach strukturierten Form vor. Normalisierungen, wie aus dem Datenbankentwurf bekannt[5], wurden nicht vorgenommen, was zwar teilweise zu den aus diesem Bereich bekannten Problemen führt[6], für die aktuelle Anwendung aber eher irrelevant ist.

Im Gegenteil, die lineare und flache Struktur der Daten bringt eher noch Vereinfachungen mit sich, da z.B. bei der Erstellung des Suchindex ein einfaches Durchlaufen der Datei ausreicht und keine komplizierten Verknüpfungen verwaltet werden müssen.

[5] siehe z.B. http://de.wikipedia.org/wiki/Normalisierung_(Datenbank)

[6] Vor allem natürlich zu einem gewissen Maß an Redundanz, z.B. bzgl. der Autoren- oder Konferenznamen.

2.3 Statistische Auswertung

2.3.1 Das Programm `dblpanalyze`

Neben dem Format, in welchem die Daten vorliegen, spielen insb. die genau Art und Menge der durch die Suche zugänglich zu machenden Informationen eine wichtige Rolle für die Implementierung der Suchanwendung.

Das Programm `dblpanalyze` wurde extra zum Zweck der automatischen Analyse des DBLP–Datenbestandes geschrieben. Es parst die Datei `dblp.xml` und erstellt dabei verschiedene Statistiken.

Aufruf

Das Programm akzeptiert drei Parameter, deren Angabe jedoch nur nötig wird, falls die eingebauten Standardwerte nicht passend sein sollten. Die Parameter im Einzelnen:

- `-i` („Input“): Durch Angabe von `-i <dateiname>` kann bestimmt werden, aus welcher DBLP–XML–Datei die Daten gelesen werden sollen bzw. wo sich diese befindet. Als Standardwert vorgegeben ist „dblp.xml“, d.h. die Eingabedaten werden aus der Datei `dblp.xml` im aktuellen Verzeichnis gelesen.

- `-q` („Quiet“): Durch Angabe von `-q` können die Ausgaben des Programms eingeschränkt werden. Normalerweise wird während des Analysierens für alle 1000 gefundenen Publikationen eine Fortschrittsmeldung ausgegeben. Ist dieser Parameter angegeben, werden diese Meldungen nicht angezeigt.

- `-?`: Der Parameter `-?` liefert eine Syntaxhilfe und ein paar kurze Informationen zum Programm.

Wichtig: Die Datei `dblp.dtd` muss sich im aktuellen Verzeichnis befinden, da sie von `dblp.xml` in dieser Weise referenziert wird!

Ausgaben

Als Ausgabe liefert `dblpanalyze` Statistiken über die Gesamtzahl der Publikationen, die Anzahl der Vorkommen der verschiedenen Publikations- und Feldtypen sowie genauere Angaben über das Vorkommen der einzelnen Feldtypen in den Publikationen–Datensätzen.

2.3.2 Ergebnisse der Analyse

Das DBLP erfasste zum Zeitpunkt der ersten Auswertung mit dem Datenbestand vom 23. Feb. 2007 insgesamt 860157 Publikationen.[7]

Diese teilen sich auf in verschiedene Typen, wie z.B. Zeitschriften-Artikel, Bücher, Papiere zu Konferenzen, etc. Die Anzahl der erfassten Publikationen der jeweiligen Typen im DBLP ist sehr unterschiedlich (siehe Tabelle 2.1).

Tabelle 2.1: Publikationstypen im DBLP, geordnet nach Anzahl

Typ	Anzahl	ca. Anteil in %
inproceedings	527435	61%
article	310097	36%
www	10179	1.2%
proceedings	8599	1%
incollection	2530	0.29%
book	1225	0.14%
phdthesis	85	$< 0.01\%$
mastersthesis	7	$< 0.01\%$
Summe	860157	100%

Auch hinsichtlich der Anzahl der Vorkommen der einzelnen Feldtypen, mittels derer die eigentlichen Informationen zu den Publikationen abgespeichert sind, gibt es große Unterschiede: Manche, wie „title“, „author“ oder auch „year“ sind in praktisch allen Datensätzen mindestens einmal vorhanden; andere wiederum wie „address“ oder „chapter“ treten dagegen nur extrem selten auf (siehe Tabelle 2.2).

Dies kann bei der Entwicklung der Suche insofern relevant sein, als dass man evtl. Für einzelne Feldtypen davon absehen könnte, sie in den Suchindex mit aufzunehmen. Daten für das Feld „chapter“ sind z.B. lediglich bei zwei Datensätzen überhaupt vorhanden. Selbst wenn also ein Nutzer das Kapitel als Suchkriterium angibt ist die Wahrscheinlichkeit, dass die Information darüber für die gesuchte Publikation vorhanden ist und diese damit überhaupt gefunden werden kann, extrem gering.

Abhängig von der genauen Implementierung der Sucheanwendung lässt sich durch Weglassen von Feldern u.U. Speicherplatz sparen oder die Zeit zur Erstellung des Suchindex oder auch die Dauer der Suche selbst verkürzen.

[7] Juli 2007: Die Zahlen haben sich seither weiter erhöht. Allerdings dienten die hier präsentierten Daten als Grundlage für die weitere Planung und die relevanten Gegebenheiten, wie z.B. die relativen Verhältnisse der Feldhäufigkeiten zueinander oder die Vorkommen pro Publikation, haben sich auch nicht signifikant verändert.

Neben der absoluten Anzahl der Vorkommen der Feldtypen ist insb. noch relevant, wie oft ein Feldtyp innerhalb eines Datensatzes vorkommen kann. Hier ist zu sehen, dass es — mit der einzigen Ausnahme von „title“ — offensichtlich keinen Feldtyp gibt, der in *allen* Datensätzen vorkommt. Andererseits gibt es verschiedene Feldtypen, die in einigen Datensätzen auch mehrfach enthalten sein können. Als Extrembeispiele fällt hier „cite“ ins Auge, das 741 Mal in einem einzigen Datensatz anzutreffen war; außerdem scheint eine der Publikationen von 115 Autoren geschrieben worden zu sein.

Diese variable Anzahl von Datenzuordnungen pro Datensatz muss also ebenfalls von der Suchmaschine bzw. dem Suchindex unterstützt werden, damit die Datenbasis korrekt abgebildet und damit auch korrekt durchsucht werden kann.

Tabelle 2.2: Feldtypen für Publikationen, geordnet nach Anzahl

Feldtyp	Anzahl	Enth. in…	Nicht in…	Min.…	Max.…
author	2059936	845896	14261	0	115
title	860157	860157	0	1	1
url	858158	858155	2002	0	2
year	849991	849991	10166	0	1
pages	795175	795171	64986	0	2
booktitle	538397	538397	321760	0	1
ee	519803	513873	346284	0	3
crossref	407642	407642	452515	0	1
volume	314943	314943	545214	0	1
journal	309880	309880	550277	0	1
number	288540	288540	571617	0	1
cite	172401	8212	851945	0	741
editor	17594	6986	853171	0	27
cdrom	14793	14236	845921	0	2
publisher	9930	9911	850246	0	2
isbn	8642	8619	851538	0	3
series	5694	5692	854465	0	2
month	2480	2480	857677	0	1
note	196	196	859961	0	1
school	92	92	860065	0	1
address	4	4	860153	0	1
chapter	2	2	860155	0	1
		…Datens.	…Datens.	…Mal/DS	…Mal/DS

2.3.3 Art der Dateninhalte

Schlussendlich spielt auch noch die genaue Art bzw. das Format der Daten in den einzelnen Feldern eine Rolle. Je nachdem, welche Inhalte hier wie vorliegen, kann es erforderlich oder sinnvoll sein, diese bei der Erstellung des Suchindex unterschiedlich zu behandeln.

So kann es z.B. Sinn machen, den Inhalt von Feldern wie „title“ oder auch „author“ nicht als Ganzes zu betrachten, sondern wiederum in kleinere Einheiten zu unterteilen, da anzunehmen ist, dass natürlich nicht immer der gesamte Inhalt bei einer Suchabfrage angegeben wird. Felder wie z.B. „url“ weiter zu unterteilen macht dagegen wahrscheinlich keinen Sinn, da es sich dort weitestgehend um eine geschlossene Einheit handelt.

Falls es möglich wäre[8], bei der Speicherung der Suchterme im Index Datentypen anzugeben, könnte sich evtl. bei der Behandlung der Jahreszahlen, Kapitel–Angaben, etc. durch Behandlung als Zahlen (statt Text) Speicherplatz sparen lassen.

Da diese Aspekte aber wiederum zu einem großen Teil von der genauen Implementierung der Suchanwendung bzw. den dabei verfügbaren Möglichkeiten abhängen, werden sie erst in Abschnitt 4.3.2 genauer behandelt.

[8] Was, zumindest bei den im Rahmen dieser Arbeit untersuchten Suchmaschinen, allerdings nicht der Fall ist.

Kapitel 3

Vorüberlegungen

Nachdem die vorhandene Ausgangssituation und die zu erreichenden Ziele geklärt sind, kann jetzt die Planung der eigentlichen Implementierung der neuen Suchanwendung erfolgen.

Die vielleicht wichtigste hier zu treffende Entscheidung bezieht sich auf die Wahl der als Grundlage der Anwendung zu benutzenden Suchmaschine. Da es aus einer Reihe von Gründen sinnvoll ist, auf einer bereits existierenden, ausgereiften Suchmaschine aufzusetzen, ist es erforderlich, einen Überblick über die Leistungen und Features bestehender Projekte zu bekommen und anhand der speziellen, für die DBLP–Suche relevanten Kriterien, eines der Projekte auszuwählen.

Ein weiterer Aspekt, der in der Planung eine Rolle spielt, ist die Auswahl der für die Suchanwendung zu verwendenden Programmiersprache. Neben einer Reihe von eigenständigen Punkten hängt die Wahl auch mit der Wahl der Basis–Suchmaschine zusammen, da diese natürlich mit der gewählten Sprache benutzbar sein muss.

Gedanken zur Bedienung und Oberfläche insb. der eigentlichen Suche bilden den dritten Abschnitt. Da dies der Teil der Suchanwendung ist, mit der der Nutzer direkt konfrontiert wird, gilt es hier eine angemessene Umsetzung zu finden. Außerdem spielen auch hier Verflechtungen mit Suchmaschine und Programmiersprache eine gewisse Rolle.

3.1 Auswahl der Suchmaschine

Die Implementierung einer schlichten Volltextsuche nach einer einfachen Zeichenkette und ohne große Berücksichtigung von Performance oder Flexibilität ist noch ein verhältnismäßig einfaches Unterfangen. Möchte man aber weitere Features zur Verfügung stellen, sei es nun die Suche in strukturierten Daten mit

mehreren Feldern, erweiterte Abfragemöglichkeiten wie Ähnlichkeitssuche oder Suche mittels Platzhaltern oder auch nur eine angemessene Performance auch bei steigender Datenmenge, so erkennt man schnell, dass hierfür erheblich mehr Aufwand und gutes Hintergrundwissen benötigt werden.

Aus diesem Grund macht es Sinn, anstelle das Rad neu zu erfinden beim Erstellen einer eigenen, spezialisierten Suchanwendung stattdessen auf der Basis einer bereits existierenden, ausgereiften, allgemeinen Suchmaschine aufzusetzen. Man erspart sich hierbei nicht nur die Arbeit, die die Entwicklung einer solchen Suchengine an sich mit sich bringt; oft stehen den entsprechenden Projekten auch eine große Anzahl von Entwicklern mit teilweise breiterem als dem eigenen Fachwissen zur Verfügung, so dass man selbst von der kontinuierlichen Weiterentwicklung, Bugfixes und Neuerungen profitieren kann, ohne erhebliche eigene Arbeit investieren zu müssen.

Im Rahmen dieser Arbeit wurden mehrere Suchmaschinen als Grundlage für die neue DBLP–Suche in Betracht gezogen:

3.1.1 Zettair

Allgemeine Informationen

Zettair (bis Mai 2004 unter dem Namen „Lucy“ bekannt) ist eine in C geschriebene, relativ kompakte und schnelle Suchmaschine. Entwickelt wurde sie von der Search Engine Group an der RMIT University, Melbourne, Australien. Laut den Aussagen der Entwickler standen beim Entwurf der Engine vor allem Einfachheit, Geschwindigkeit und Flexibilität im Vordergrund, sowie die Fähigkeit, auch große Mengen von Text(en) gut und effizient zu handhaben.[1]

Das Programm unterstützt sowohl die Indexierung von reinen Texten und HTML–Dateien als auch Textsammlungen im TREC–Format[2]. Zettair unterstützt Boolean–Queries nach einzelnen Wörtern und Phrasen mit UND- und ODER–Operatoren und Klammerung (auch geschachtelt). Ein Ranking der Ergebnisse wird ebenfalls durchgeführt.

Zettair stellt ein ausführbares Programm zur Verfügung, welches, mittels einer Kommandozeilen–Schnittstelle, sowohl das Erstellen des Index als auch die Suche darin ermöglicht. Des weiteren kann die Suchmaschine mittels einer modular

[1] vgl. [Zett06].

[2] Trotz intensiver Suche ist es mir nicht gelungen, irgendeine Beschreibung oder Spezifikation dieses Formates zu erhalten. Für meine Versuche war ich deshalb auf die sehr spärliche, mit Zettair mitgelieferte Dokumentation zu diesem Thema und die eigene Analyse des dortigen Beispieltextes angewiesen.

aufgebauten C–API leicht[3] in andere Anwendungen integriert werden.

Laut Website wurde die Suchmaschine unter Linux, FreeBSD, Mac OS X (Darwin), Solaris, Win32 (Windows 95, 98, NT, etc.) und der Cygwin–Umgebung[4] erfolgreich getestet und sollte in den meisten POSIX–kompatiblen Umgebungen funktionieren.

Zettair steht unter einer BSD-ähnlichen Lizenz[5] und kann unter `http://www.seg.rmit.edu.au/zettair/` heruntergeladen werden.

Bewertung

Zettair wurde von Michael Ley als eine mögliche Ausgangsbasis für die DBLP–Suche vorgeschlagen. Um die Eignung für diesen Zweck genauer einschätzen zu können, wurden einige Tests unternommen:

Unter Verwendung des mitgelieferten Programms `zet` wurde aus der unmodifizierten Datei `dblp.xml` ein Zettair–Suchindex generiert. Auf der Basis dieses Suchindex wurden dann einige Beispiel–Suchabfragen abgesetzt. Aufgrund der Tatsache, dass der Index nur über eine einzige Datei generiert wurde und Suchabfragen demzufolge natürlich auch immer nur diese eine Datei als Ergebnis zurückliefern, ist dieser Test zwar alles andere als praxisnah; allerdings liefert er zumindest ein Beispiel für die Geschwindigkeit von Zettair: Die komplette Indexerstellung aus der 385 MiB großen Datei dauerte im Durchschnitt nur 53 Sekunden. Abfragen benötigen, je nach Komplexität und Anzahl der angegebenen Suchterme, wenige Millisekunden bis wenige Hundertstelsekunden.

Als nächstes wurde unter Verwendung des speziell dafür geschriebenen C^{++}–Programms `dblp2trec` die Datei `dblp.xml` in das TREC–Format umgewandelt. Die entstandene Datei wurde wiederum mittels des Programms `zet` indexiert. Auch hier zeigte sich wieder die hohe Geschwindigkeit von Zettair, wobei die Generierung des Index hier sogar nur durchschnittlich 45 Sekunden dauerte (allerdings ist dabei zu beachten, dass die entstandene TREC–Datei mit ca. 220 MiB auch signifikant kleiner als die originale `dblp.xml` ist, so dass der Datendurchsatz an sich hier etwas geringer ist). Die Geschwindigkeit der Abfragen wurde nicht merkbar beeinflusst.

Klar ist natürlich, dass für eine wirkliche Bewertung der Geschwindigkeit mehr und umfangreichere Tests durchgeführt werden müssten; für eine anfängliche Beurteilung sollten die durchgeführten Versuche aber ausreichend sein.

[3] Eine zum Testen geschriebene Mini–Beispielsuche umfasst alles in allem nur 25 Zeilen.

[4] Eine Linux–artige Umgebung für Windows, siehe `http://www.cygwin.com/`

[5] Der der Distribution beiliegende README.TXT spricht zwar von der GPL, aber auf der Website wird auf eine „BSD–style license“ verwiesen und die der Distribution beiliegende Lizenz in `copying.html` ist definitiv ebenfalls nicht die GPL.

Um die Handhabbarkeit der Zettair–C–API beurteilen zu können, wurde letztlich noch eine kleine Beispielsuche geschrieben. Die Einbindung der Zettair–Engine in eigene Programme gestaltet sich sehr einfach lediglich durch Einbinden einer Include–Datei und Verlinken mit der Bibliothek `libzet`. Laden des Index sowie Absetzen der Suchabfrage und Rückgabe der Ergebnisse lassen sich jeweils mit einem einzigen Befehl erledigen, so dass auch die Benutzung der Engine keine Schwierigkeiten bereitet.[6]

Wie die obigen Tests zeigen, wird Zettair den selbstgestellten Anforderungen an Geschwindigkeit und Einfachheit durchaus gerecht und kann in diesem Bereich einige Pluspunkte verbuchen.

Als Unterbau einer Suche für das DBLP eignet sich Zettair allerdings nur sehr bedingt. Aufgrund der Art und insb. der Struktur der hier vorliegenden Daten unterscheidet sich eine Suche im DBLP doch sehr von einer Suche in einer Sammlung „normaler“, weitgehend unstrukturierter Textdokumente. Während dort die Suchgrundlage aus einer großen Menge von im Prinzip gleichberechtigten Termen besteht, sind die Daten im DBLP sowohl semantisch als auch syntaktisch gruppiert (→ 2.2). Um die Möglichkeiten des DBLP voll nutzen zu können, sollte eine Suchmaschine diese Strukturierung berücksichtigen und unterstützen können.

3.1.2 MG — Managing Gigabytes

Allgemeine Informationen

Die Suchmaschine MG, was als Abkürzung für „Managing Gigabytes“ steht, wurde im Zuge des gleichnamigen Buches „Managing Gigabytes: Compressing and Indexing Documents and Images“ von Ian H. Witten, Alistair Moffat und Timothy C. Bell [WMB94] entwickelt und ist dort ausführlich beschrieben.[7]

Das Programm unterstützt Dateien in den Formaten HTML, ASCII (reiner Text), MAIL, BibTEX und TEX, auch in komprimierter Form als .Z und .gz–Archive. MG unterstützt Boolean–Queries nach einzelnen Wörtern und Phrasen mit UND- und ODER–Operatoren und Klammerung (auch geschachtelt). Ein Ranking der Ergebnisse wird ebenfalls durchgeführt.

MG stellt darüber hinaus noch weitere Funktionalität, u.A. zur Komprimierung von Bilddaten und eine graphische Nutzeroberfläche für das X–Window–

[6] Bei den Tests mit den mitgelieferten — jedoch lokal kompilierten — Programmen traten allerdings in unregelmäßigen Fällen Abstürze bei der Indexerstellung auf. Ob es sich dabei um Systeminkompatibilitäten, Probleme in der Demo–Anwendung oder Probleme der Bibliothek an sich handelte, konnte nicht ermittelt werden.

[7] Eine schöne Übersicht der MG–Features findet sich weiterhin auch unter http://www.ncsi.iisc.ernet.in/raja/netlis/wise/mg/mainmg.html

System bereit.

Eine „offizielle“ Liste der unterstützten Betriebssystem war weder in der Dokumentation noch auf den Webseiten zu finden; laut diverser Aussagen und Hinweisen in der Dokumentation und anderen Stellen scheint MG allerdings unter den meisten Betriebssystemen wie Linux, Windows, Mac und diversen Unix–Derivaten zu funktionieren. Insb. wenn man beachtet, dass die entsprechenden Informationen teilweise bereits bis zu zehn Jahre alt sind, kann man davon ausgehen, dass mittlerweile für die meisten einigermaßen verbreiteten Systeme entsprechende Patches oder Anleitungen existieren.

MG steht unter der GPL und kann unter http://www.cs.mu.oz.au/mg/ oder (in einer im Bezug auf die Dokumentation etwas erweiterten Version) unter http://www.math.utah.edu/pub/mg/ heruntergeladen werden.

Bewertung

Wie schon bei Zettair handelt es sich auch bei MG primär um eine Suchmaschine zur Volltextsuche, was sie aus den gleichen Gründen für eine Verwendung im Rahmen dieser Arbeit als eher ungeeignet erscheinen lässt. Deshalb wurden aus Zeit-Gründen auch keine weitergehenden Versuche mit dieser Engine durchgeführt.

Die bisherige Erweiterte Suche für die DBLP setzt zwar auf MG als Grundlage auf, dieser Ansatz wurde jedoch für diese Arbeit nicht weiter verfolgt.

3.1.3 Lucene

Allgemeine Informationen

Lucene ist eine in Java geschriebene Bibliothek bzw. API, die Anwendungen die Möglichkeit gibt, eigene Suchen zu implementieren. Lucene wurde im Rahmen des Apache–Projekts[8] entwickelt und bietet eine sehr große Anzahl von Features.

Lucene unterstützt alle in Abschnitt 1.4 aufgeführten Suchtypen. Darüber hinaus kann Lucene, im Gegensatz zu den bisher erwähnten Engines, auch direkt mit strukturierten Daten mit mehreren Datenfeldern umgehen und diese korrekt indexieren.

Lucene stellt jeweils mehrere Klassen für die Token–Erzeugung, Stemming und andere Filterungen zur Verfügung.[9] Klassen zur Unterstützung von WordNet[10] ermöglichen u.A. die Suche nach Synonymen.

[8] siehe http://www.apache.org/.

[9] Zu den einzelnen Schritten der Index–Erzeugung siehe Abschnitt 4.1.3.

[10] Eine umfangreiche, lexikalische Datenbank für die Englische Sprache, siehe http://

Da Lucene in Java geschrieben ist, ist die Bibliothek unter allen Betriebssystemen, für die eine Java–Laufzeitumgebung zur Verfügung steht — insb. also auch Windows und Linux — benutzbar.

Lucene steht unter der Apache License v2.0 und ist auf der Homepage unter `http://lucene.apache.org/` verfügbar.

Bewertung

Lucene wurde im Rahmen einer Suche nach weiteren Optionen für eine Basis–Suchmaschine gefunden und fiel aufgrund der vielen Features und der Unterstützung für strukturierte Daten sofort positiv auf.

In der Lucene–Distribution mitgeliefert wird eine Demo–Anwendung, die allerdings auf die Indexierung und Suche in einer Menge von Text- oder HTML–Dateien ausgerichtet ist und diese, eher wieder im Rahmen einer allgemeinen Volltextsuche, aufbereitet. Für einen Test mit den DBLP–Daten auf Basis der Datei `dblp.xml` ist diese Anwendung deshalb nicht sehr gut geeignet. Aus diesem Grund wurde eine eigene kleine Beispielanwendung erstellt, die zum Testen der API, der Performance und der allgemeinen Eignung verwendet wurde.

Die Benutzung der Bibliothek in eigenen Anwendungen erweist sich als sehr einfach; nach der Java–üblichen Einbeziehung der Lucene–Klassen in den Class-Path stehen die entsprechenden Klassen dem eigenen Programm zur Verfügung.

Die Erstellung eines Index erweist sich ebenfalls als sehr einfach: Ein Objekt der Klasse `IndexWriter` wird mit einem passenden `Analyzer` und evtl. weiteren Optionen konfiguriert; die zu indexierenden Dokumente werden eingelesen, `Document`–Objekte dafür erstellt und für die enthaltenen Daten werden, je nach Wunsch, entsprechende `Field`–Objekte gebaut und dem `Document` hinzugefügt. Die `Document`–Objekte werden dem Index hinzugefügt und am Ende wird dieser noch komprimiert.[11]

Nach den vorliegenden Erfahrungen mit dieser Engine stellt Lucene aufgrund der vielen Features, der einfachen Anwendbarkeit und der guten Performance eine sehr ansprechende Option zur Verwendung in der neuen DBLP–Suche dar. Die schlussendliche Entscheidung *gegen* Lucene resultiert jedoch daher, dass Michael Ley sich aufgrund bisheriger Erfahrungen mit hohen Systemanforderungen und teilweise Kompatibilitätsproblemen zwischen verschiedenen Java–Versionen gegen eine Java–Applikation ausgesprochen hat.

`wordnet.princeton.edu/`.

[11] Eine etwas ausführlichere Beschreibung findet sich in Abschnitt 4.3.2; auch wenn das dort beschriebene Programm `dblp2index` auf der CLucene–Engine aufsetzt ist der Ablauf doch weitestgehend gleich.

3.1.4 CLucene

Allgemeine Informationen

CLucene ist die C^{++}–Portierung der Lucene–Engine. Wie auch bei dieser handelt es sich um eine Bibliothek bzw. API zur Einbindung in eigene Applikationen. CLucene unterstützt alle wichtigen Lucene–Features, insb. also auch alle dort implementierten Suchtypen und die Behandlung und Indexierung von strukturierten Daten.

Viele der nicht direkt zum Kernbereich von Lucene gehörenden Features wurden allerdings (zumindest bisher noch) nicht auf CLucene portiert. Darunter fallen z.B. die speziellen Analyzer für eine Reihe von Sprachen, Regular–Expression–Queries und auch die WordNet–Unterstützung.

Laut Dokumentation läuft CLucene unter Linux, Mac OS X (Darwin), Win32 (Windows 95, 98, NT, etc.) und der Cygwin–Umgebung mit MingW[12].

CLucene steht ebenfalls unter der Apache License v2.0 oder alternativ unter der LGPL (GNU Library or Lesser General Public License). Heruntergeladen werden kann sie von der Homepage http://clucene.sourceforge.net/index.php/Main_Page.

Bewertung

Aufgrund der Tatsache, dass es sich bei CLucene um eine Portierung von Lucene handelt, gestaltet sich die Bewertung der Engine sehr ähnlich. Zwar werden, wie bereits erwähnt, nicht alle Features der Original–Lucene–Distribution unterstützt; für die neue DBLP–Suche wirklich interessante oder gar benötigte Features fehlen aber nicht.

Lediglich das Fehlen der WordNet–Unterstützung ist ein bisschen schade, da eine Synonym–Suche dadurch zumindest nicht ohne eigenen Programmieraufwand realisiert werden kann.[13]

Die API–Definition und damit die Programmierung an sich unterscheiden sich nur unwesentlich. Unterschiede sind fast nur durch die andere Syntax bzw. Konventionen von C^{++} gegenüber Java bestimmt.

Wie auch in der CLucene–Dokumentation und auf der CLucene–Website erwähnt[14] sollte aufgrund der Tatsache, dass es sich bei C^{++} um eine kompilierte

[12] Eine Sammlung von frei verfügbaren Bibliotheken und Header–Dateien, siehe http://www.mingw.org/

[13] Einige kleinere Tests mit der WordNet–API haben allerdings gezeigt, dass auch dieser Aufwand sich in Grenzen halten würde.

[14] siehe [CLben]

und direkt vom Prozessor ausgeführte Sprache handelt, CLucene eigentlich eine bessere Performance vorweisen können. Interessanterweise konnten diese Ergebnisse bei den durchgeführten Tests allerdings nicht bestätigt werden; die benötigte Zeit zur Erstellung eines Index mit CLucene und Lucene halten sich ungefähr die Waage.[15]

Nichtsdestotrotz sprechen insgesamt eine Reihe von Vorteilen für CLucene:

- Die Bibliothek inkl. Quellcode ist frei verfügbar.
- Die Anwendung bzw. Einbindung in eigene Programme ist einfach möglich.
- CLucene bietet eine gute Performance sowohl bei Indexierung als auch bei der Suche.
- Es werden viele interessante und nützliche Suchtypen bereitgestellt.
- Unterstützung von strukturierter Suche mit Feldern ist gegeben.

CLucene stellt damit von den untersuchten Suchmaschinen die beste Wahl dar und dient daher als Grundlage für die neue DBLP–Suche.

3.2 Auswahl der Programmiersprache

In der Praxis der Softwareentwicklung folgen einzelne Schritte meist nicht streng linear aufeinander, sondern laufen parallel. Oft beeinflussen sich verschiedene Aspekte und Überlegungen gegenseitig und führen damit automatisch zu einem solchen Ablauf. Auch die Planung bzgl. der zu verwendenden Programmiersprache erfolgte so während der Evaluierung der Suchmaschinen.

Das vielleicht wichtigste Kriterium in diesem Bereich ist so wenig objektiv, wie es relevant ist: Da weder die Zeit dafür vorhanden ist, noch das eigentliche Thema der Arbeit im Lernen einer neuen Programmiersprache besteht, kommen nur solche Sprachen in Frage, die vom Entwickler bereits ausreichend beherrscht werden :-)

Aus diesem Grund beschränkt sich die Grundauswahl der möglichen Sprachen auf Java, Perl, PHP, C und C^{++}. Bei der Entscheidung für eine dieser Sprache hingegen spielen allerdings durchaus noch weitere objektiv zu betrachtende Punkte eine Rolle.

[15] Dauer Indexerstellung Lucene: ca. 26 Minuten; Dauer Indexerstellung CLucene: ebenfalls grob um die 26 Minuten, teilweise sogar länger; genauere Angaben zur Performance siehe Abschnitt 4.3.3.

3.2.1 Java

Wie bereits bei der Wahl zur Suchmaschine erwähnt (→ 3.1.3) kommt eine Java–Anwendung, zumindest für den Einsatz in der Praxis, aufgrund insb. von Performance- und Kompatibilitätsproblemen nicht in Frage. Die Entscheidung für eine Implementierung in dieser Sprache hätte also dazu geführt, dass die entstandene Applikation lediglich in Bezug auf diese Diplomarbeit ihre Daseinsberechtigung gehabt hätte.

An sich bietet die Sprache mit Objektorientiertheit, einer großen Anzahl von verfügbaren Bibliotheken, Plattformunabhängigkeit, etc. eine Reihe von Pluspunkten. Aufgrund der oben angeführten Einschränkung wurde jedoch gegen die Benutzung von Java entschieden, da eine von vorneherein nur zum Zweck der „Pflichterfüllung“ durchgeführte Implementierung als insgesamt nur wenig sinnvoll erschien.

3.2.2 Perl

Die Skriptsprache Perl wird noch verhältnismäßig oft für Webanwendungen verwendet und es existiert auch eine Lucene–Portierung für diese Sprache[16], so dass auch eine Implementierung der neuen Suche in Perl möglich wäre.

Die Entscheidung gegen Perl erfolgte zum größten Teil aufgrund der mangelnden Erfahrung bzgl. der objektorientierten Aspekte dieser Sprache. Zwar handelt es sich bei der neuen DBLP–Suche insgesamt immer noch um eine verhältnismäßig kleine Anwendung, jedoch sollte eine Entwicklung nichtsdestotrotz nach OO–Prinzipien erfolgen, um die daraus resultierenden Vorteile wie Modularität, leichtere Wartbarkeit, etc. nutzen zu können.

Allerdings gab es weiterhin auch gewisse Befürchtungen hinsichtlich der Performance, denn bei Perl handelt es sich um eine Interpretersprache, für die bei jedem Aufruf erst einmal der Interpreter gestartet werden muss.[17]

Außerdem muss auf dem Serversystem natürlich noch die Perl–Umgebung installiert sein, was einen erhöhten Installationsaufwand bedingt, auch wenn davon auszugehen ist, dass zumindest auf den meisten Unix/Linux–Systemen bereits eine Perl–Installation vorhanden sein dürfte.

[16] PLucene, siehe http://search.cpan.org/~tmtm/Plucene-1.25/lib/Plucene.pm
[17] vgl. z.B. [WPpp].

3.2.3 PHP

PHP wurde ursprünglich als Sprache speziell für die Entwicklung von Webanwendungen entwickelt, was an sich natürlich ein Pluspunkt für die Umsetzung der Web–basierten Suchapplikation wäre. Mit dem Modul Zend_Search_Lucene[18] steht auch die Lucene–Funktionalität zur Verfügung.

Hauptgrund gegen die Benutzung von PHP war auch hier wieder die mangelnde Erfahrung mit den objektorientierten Features. Weiterhin war fraglich, ob sich PHP für die Entwicklung der nicht–Web–orientieren Programme z.B. zur Indexerstellung eignen würde, da auch hier die Erfahrungen mit solchen „eigenständigen“ PHP–Applikationen fehlen.

3.2.4 C und C^{++}

Die Wahl für die zu verwendende Programmiersprache fiel schließlich auf C^{++}, insb. da auch Michael Ley als Administrator des DBLP sich für diese Sprache ausgesprochen hatte.

Wie oben bereits erwähnt existiert ein nativer Lucene–Port für C^{++}; die Sprache ist objektorientiert und wird kompiliert, benötigt also (zur Laufzeit) keine weiteren installierten Interpreter, Laufzeitumgebungen oder spezielle Server–Module[19]. Außerdem führt dies zu im Normalfall besserer Performance und geringerem Ressourcenverbrauch gegenüber interpretierten Sprachen bzw. Sprachen mit eigener Umgebung.

3.2.5 Entwicklung in mehreren Sprachen

Da das gesamte Anwendungs–„Paket“ aus mehreren Einzelprogrammen besteht (s.u.), wäre es theoretisch auch möglich gewesen, für die einzelnen Programme unterschiedliche Sprachen zu benutzen, z.B. C^{++} für das Programm zur Indexerstellung aber PHP für die Webanwendung zur eigentlichen Suche.

Da es hierfür aber notwendig gewesen wäre, die verschiedenen Bibliotheken aus verschiedenen Sprachen heraus anzusprechen, was sich u.U. als schwierig erweist, wurde davon abgesehen. Außerdem hätte das evtl. erfordert, weiteren zusätzliche Installationsaufwand (Installation mehrerer Interpreter bzw. Laufzeitumgebungen, Server–Module, o.Ä.) zu treiben.

[18] siehe http://devzone.zend.com/node/view/id/91

[19] Abgesehen von der CGI–Unterstützung.

3.3 Überlegungen zur Nutzeroberfläche

Das Frontend einer Anwendung ist im Allgemeinen der erste und aus Nutzersicht meist auch wichtigste Teil einer Applikation, mit dem der Anwender konfrontiert wird. Aus diesem Grund macht es Sinn, auch hierzu einige Überlegungen anzustellen, um eine adäquate Umsetzung zu erzielen.

Vergleicht man die Nutzeroberflächen heutiger Suchmaschinen[20], so erkennt man, dass es vor allem zwei Varianten zur Eingabe der Suchbegriffe gibt:

(1) *Query/Zeichenkette:* Die Nutzeroberfläche besteht aus einer einzigen Eingabezeile, in die der Nutzer seine Abfrage, kodiert als Zeichenkette eingibt.

(2) *Formular:* Je nach Art der Daten gibt es mehrere Eingabefelder, in welche der Nutzer je nach Wunsch entsprechende Suchbegriffe einträgt.

Neben diesen Hauptformen gibt es auch noch speziellere Arten von Oberflächen[21], die aber generell bisher eher wenig verbreitet sind.

3.3.1 Query/Zeichenkette

Vorteile dieser Methode sind natürlich ihre Einfachheit, nicht nur in der Programmierung (zumindest soweit es die Oberfläche betrifft), sondern auch — zumindest teilweise — bei der Anwendung.

Die Lucene–API stellt direkt eine Klasse für das Parsen einer als Zeichenkette eingegebenen Abfrage bereit, so dass hier keine eigene Programmierung erforderlich ist. Der Nutzer andererseits kann im einfachsten Fall einfach einen oder mehrere Suchbegriffe eingeben und auf diese Weise eine Abfrage absetzen.

Für die Einbindung in bestehende Webseiten ist weiterhin zu sagen, dass eine Suchmaske dieser Form nur sehr wenig Platz einnimmt und nach Wunsch normalerweise auf jeder Seite leicht zusätzlich untergebracht werden kann.

Allerdings hat diese Methode auch Nachteile: Einmal ist natürlich ein, je nach verwendeter Abfrage–Sprache evtl. aufwändiges, Parsen des Queries erforderlich. Da diese Arbeit aber bereits von der Lucene–API abgenommen wird, stellt dieser Punkt hier kein Problem dar.

Wichtiger ist hier wohl, dass der Nutzer Kenntnisse der Syntax der verwendeten Abfrage–Sprache haben muss, will er komplexere Abfragen benutzen. Zwar

[20] Hier beschränkt auf Suchmaschinen zur Suche nach Text bzw. textuell abgelegten Daten; für andere Daten, z.B. Bilddateien, sind u.U. andere Vorgehensweisen wesentlich sinnvoller oder sogar erforderlich.

[21] vgl. z.B. [BR+99], Kapitel 10.

ist die Lucene–Query–Syntax verhältnismäßig einfach und aufgrund des Themengebietes des DBLP ist davon auszugehen, dass die meisten Nutzer zumindest gewisse Erfahrung in diesem Bereich vorweisen können; nichtsdestotrotz sollte man diesen Punkt aber nicht ganz vernachlässigen.

3.3.2 Formular

Bei dieser Methode ist z.B. Für jedes Datenfeld eine eigene Eingabemöglichkeit vorhanden.[22]

Bzgl. der Programmierung bietet dies sowohl Vor- als auch Nachteile. Von Vorteil ist einerseits, dass ein evtl. aufwändiges und möglicherweise fehlerbehaftetes Parsen der Abfrage entfällt. Eine evtl. Korrektheitsüberprüfung der Eingaben wird dadurch erleichtert, dass die gewünschte Art der Daten in jedem Feld bekannt ist.

Außerdem können aus diesem Grund auch Steuerungsmöglichkeiten für die Suche, wie z.B. die Einstellung von Ähnlichkeitssuche oder exakter Suche, direkt für jedes Feld als zusätzliches Oberflächen–Steuerelement bereitgestellt werden. Dies erleichtert natürlich die Anwendung solcher fortgeschrittener Features für den weniger versierten Nutzer.

Nachteilig gestaltet sich, dass jetzt in der Oberflächen–Programmierung jedes Feld bzw. Kontrollelement auch einzeln behandelt werden muss. Des weiteren müssen die einzelnen Eingaben dann zu einer, der API der Suchmaschine entsprechender, Abfrage zusammengesetzt werden.

Außerdem verbraucht ein solches Formular, insb. bei einer großen Anzahl von Datenfeldern und Steuermöglichkeiten natürlich wesentlich mehr Platz als ein einfaches Eingabefeld.

3.3.3 Entscheidung

Letztendlich viel die Entscheidung für die Eingabemaske auf die Benutzung eines einfachen Eingabefeldes. Da praktisch die gesamte Arbeit zum Parsen und Auswerten der Query–Zeichenkette bereits von CLucene erledigt wird, ergibt sich hier nur ein geringer Aufwand. Außerdem ermöglicht es diese Variante, eine Suche nach Titel(–Stichworten) auf einfachste Weise nur durch Eingabe des Suchterms bzw. der Suchterme und Drücken der Eingabetaste zu erreichen. Suchen nach anderen Datenfeldern sind ebenfalls mit nur geringem Mehraufwand (zusätzliche Angabe des Feldnamens) möglich und auch die anderen Sucharten können — bei Kenntnis der Abfragesyntax — in dieser Weise leicht angegeben werden.

[22] vgl. z.B. Erweiterte DBLP–Suche (→ 1.2.3)

Sollte später noch eine erweiterte Eingabemöglichkeit mittels Formular gewünscht werden, ist es ohne Probleme möglich, diese ebenfalls noch nachträglich zu ergänzen.

Kapitel 4

Der Suchindex

Bei verhältnismäßig geringem Datenvolumen stellt einfaches lineares Durchsuchen der verfügbaren Daten zur Suche nach bestimmten Informationen oft noch eine praktikable Option dar. Sobald aber die Menge der zu durchsuchenden Daten größer wird oder auch einfach komplexere Suchmöglichkeiten angestrebt werden, stößt diese Vorgehensweise schnell an ihre Grenzen.

Durch vorangehendes Analysieren und Aufbereiten der zu durchsuchenden Daten ist es möglich, sowohl die Optionen zur Datensuche als auch die Performance der Suche an sich erheblich zu verbessern. Die wahrscheinlich bekannteste und für diesen Zweck am häufigsten eingesetzte Datenstruktur ist der Invertierte Index[1].

In diesem Kapitel werden diese Datenstruktur, ihre Vorteile bzw. Anwendungsbereiche und die für die Erstellung eines solchen Index notwendigen Schritte beschrieben. Anhand des Lucene–Index–Formats wird die Umsetzung dieser Datenstruktur in einer konkreten Implementierung vorgestellt. Schließlich wird auf das für die DBLP–Suche geschriebene Programm `dblp2index` eingegangen und die spezielle Vorgehensweise und die speziellen Anforderungen bei der Generierung des Suchindex für die DBLP–Suche werden erklärt.

[1] Teilweise auch als „Invertierte Datei“ bezeichnet. Insb. setzen alle der oben vorgestellten Suchmaschinen auf einem Invertierten Index auf.

4.1 Der Invertierte Index

4.1.1 Problemstellung und Motivation

Grundproblem aus IR–Sicht

In der Terminologie des Information Retrieval lässt sich die Situation einer Suche nach Informationen allgemein wie folgt definieren:[2] Ausgangsbasis ist ein Datenbestand als eine Menge von Dokumenten. „Dokument“ bezieht sich hierbei auf eine beliebige Dateneinheit, auf der die Suche aufsetzen soll und die, je nach den speziellen Gegebenheiten und Erfordernissen, ganz unterschiedlich definiert sein kann. Es muss sich keineswegs immer um ganze Dateien oder Dokumente im alltäglichen Sinn handeln, sondern kann z.B. auch Absätze eines Textes, die einzelnen e–Mails in einer MBOX–Datei oder andere Arten von Datensätzen bezeichnen.

Innerhalb dieser Sammlung von Dokumenten[3] sollen jeweils diejenigen gefunden und zurückgeliefert werden, welche bestimmten Suchkriterien bzgl. ihres Inhalts entsprechen. Semantisch gesprochen geht es darum, einen Informationswunsch (engl. „information need“) des Nutzers zu befriedigen, indem Dokumente mit entsprechenden relevanten Informationen gefunden werden. Die Suchkriterien formuliert der Nutzer in einer Abfrage (engl. „query“), welche dann von der Suchanwendung ausgewertet wird.

Boolean–Suche und Grepping

Die einfachste und immer noch wohl am häufigsten eingesetzte Abfrageform ist die Boolean–Suche. Hierbei gibt der Nutzer einen oder mehrere Suchterme ein, evtl. verknüpft durch die Operatoren AND, OR und NOT. Mit letzteren wird spezifiziert, dass beide dadurch verknüpfte Terme in den Dokumenten vorkommen müssen, mindestens einer vorkommen muss bzw. ein Term nicht vorkommen darf.

Im Prinzip ließen sich Suchabfragen dieser Art einfach in der Weise auswerten, indem die Dokumente linear auf Vorkommen jedes Suchterms gescannt werden.[4] Für jeden Term enthält man so eine Menge von Dokumenten, die den Term beinhalten. Diese Mengen werden dann nur noch durch die den angegebenen Boolean–Operatoren entsprechende Mengenoperationen (Durchschnitt für AND, Vereinigung für OR, Subtraktion für NOT) verknüpft und man erhält dadurch die

[2] vgl. hier und im Folgenden z.B. [MRS07], S. 4ff.

[3] Oft auch als Corpus bezeichnet.

[4] Diese Vorgehensweise wird oft auch als „grepping“, nach dem bekannten Unix–Kommando `grep`, bezeichnet.

zur Suchabfrage passende Ergebnismenge.

Inzidenz–Matrix

Dieses Verfahren ist allerdings, insb. bei einer großen Anzahl großer Dokumente, sehr aufwändig, da jedes Dokument immer wieder komplett durchsucht werden muss. Eine Verbesserung lässt sich erreichen, indem die Dokumente bereits im Vorfeld einmal analysiert werden. Dabei erzeugt man eine Matrix, deren Spalten alle in allen Dokumenten vorkommenden Wörter bezeichnen; die Zeilen bezeichnen entsprechend alle vorhandenen Dokumente. Die Dokumente werden nun einmal durchlaufen und für jedes Wort (Spalte) wird in der Matrix mittels einer 1 notiert, ob das Wort in den Dokumenten (Zeile) vorkommt; andernfalls wird eine 0 eingetragen. Die so entstandene Matrix bezeichnet man auch als Inzidenz–Matrix[5].

Suchabfragen lassen sich nun einfach dadurch beantworten, dass man die den eingegebenen Suchtermen entsprechenden (Spalten–)Vektoren entsprechend der angegebenen Operatoren bitweise miteinander verknüpft. Der dabei entstehende Vektor hat dann eine 1 in allen Zeilen, die den Dokumenten entsprechen, die in der der Suchabfrage entsprechenden Ergebnismenge enthalten sind.

Allerdings erkennt man schnell, dass dieses Verfahren alles andere als optimal ist: Bei z.B. 1 Million Dokumenten, von denen jedes ca. 1000 Worte enthält und in denen insgesamt ca. 80.000 unterschiedliche Worte vorkommen wäre die Matrix mit $1.000.000 \times 80.000$ Einträgen riesig.[6] Da in jeder Spalte max. 1000 Einsen vorkommen können — jedes Dokument enthält höchstens 1000 (unterschiedliche) Worte — wäre sie außerdem noch zu mindestens 98,75% leer (d.h. die entsprechenden Zellen enthalten nur 0).

Ausgedünnte Inzidenz–Matrix, Invertierter Index

Die Idee ist nun, die Matrix so zu organisieren, dass nur die interessanten (d.h. Einser–Zellen) gespeichert werden. Dies geschieht in der Weise, dass einfach für jeden Term eine Liste der Dokumente, in denen er enthalten ist, verwaltet wird. Zwar wird dadurch die Struktur insgesamt etwas komplexer, der Speicheraufwand sinkt aber dramatisch.

Die entstandene Datenstruktur repräsentiert eine Zuordnung von Wörtern zu den Dokumenten, in denen diese Wörter vorkommen. Man bezeichnet diese Datenstruktur als (einfachen) Invertierten Index.

[5] Von lat. incidere = hinfallen

[6] Bei einer Repräsentation der Vektoren durch Bit–Vektoren ca. $1.000.000 \times 10.000 = 10$ Milliarden Bytes, also ca. 10 GiB!

4.1.2 Beschreibung der Datenstruktur

Grundstruktur (für Boolean–Abfragen)

Ein Invertierter Index besteht im einfachsten Fall aus einem sog. Wörterbuch (engl. „dictionary“; auch „lexicon“ oder „vocabulary“), welches alle in den indexierten Dokumenten vorkommenden Terme[7] aufführt. Jedem Term zugeordnet ist eine Liste der Dokumente (normalerweise referenziert über eine eineindeutige Dokument–ID, die z.B. beim Analysieren einfach als fortlaufende Nummer für jedes neue Dokument vergeben werden kann), in denen der Term vorkommt. Im Normalfall sind die Termeinträge alphabetisch und die Dokumentlisten nach Dokument–ID sortiert.

Tabelle 4.1: Beispiel für Einträge eines einfachen Invertierten Index

Wörterbuch	Dokumentliste
Alice	1, 2, 4
hatter	2, 3, 5, 6
mad	1, 5, 6
rabbit	3, 4, 6
white	3, 4
...	...

Einfach Boolean–Abfragen sind mit Hilfe dieser Datenstruktur ebenso leicht und schnell zu realisieren: für jeden Term in der Suchabfrage wird die Liste der Dokumente, in denen er enthalten ist, aus dem Index ermittelt. Die erhaltenen Listen bzw. Dokument–Mengen werden dann einfach wieder, entsprechend den evtl. noch angegebenen Boolean–Operatoren, zur Ergebnismenge verknüpft.

Erweiterungen (für Ranking, Phrasen–Suche, etc.)

Wie man sieht stellt dieser einfache Index für grundlegende Boolean–Abfragen bereits eine gute Datenstruktur dar. Allerdings wird für die Dokumente in der Ergebnismenge bisher keinerlei Bewertung durchgeführt; so wäre es z.B. wahrscheinlich interessant, Dokumente, in denen ein Suchterm mehrmals vorkommt, am Anfang der Ergebnisliste präsentiert zu bekommen.

[7] Bei den erfassten Einheiten handelt es sich u.U. nicht unbedingt um Wörter im Alltagssinn und insb. nicht unbedingt um die unverändert aus den Dokumenten entnommenen Begriffe; vgl. Abschnitt 4.1.3 und auch [MRS07], S. 20. Im Folgenden wird der Begriff Term für alle im Index verzeichneten Einheiten verwendet.

Um das zu erreichen, kann man nun nicht nur speichern *ob*, sondern auch *wie oft* ein Term in jedem Dokument vorkommt. Bei der Auswertung der Abfrage können diese Informationen dann in das Ranking einfließen.[8]

Will man auch Proximity- oder Phrasen–Queries ermöglichen, so sind ebenfalls weitere Daten erforderlich: Hierfür muss zusätzlich für jeden Term die Position bzw. Positionen verwaltet werden, an denen er in jedem Dokument vorkommt. Mit Hilfe dieser Zusatzdaten können dann wiederum entsprechende Abfragen realisiert werden.

4.1.3 Schritte zur Generierung

Die Generierung eines Invertierten Index aus einer Menge von Dokumenten umfasst eine Reihe von — teilweise allerdings optionalen — Schritten:[9]

(1) *Aufbereiten der Dokumente*

(2) *Token–Erzeugung*

(3) *Stemming*

(4) *Entfernen von Stopwörtern*

(5) *Erstellen der eigentlichen Datenstruktur*

Aufbereiten der Dokumente

Je nach dem, welche Daten indexiert werden sollen, in welchem Format sie vorliegen und welche Art der Suche darüber angestrebt wird, kann dieser Schritt eine Reihe von unterschiedlichen Maßnahmen umfassen:

- *Einlesen der Dokumente:* Der Zugriff auf die Dokumente muss ermöglicht werden. Je nach Format kann es sich dabei um das Behandeln von Dateien in einem Verzeichnis handeln, um das Scannen aller Seiten einer Website oder auch um das Einlesen einer MBOX–Datei mit Erkennung der einzelnen Mails.

- *Extrahieren des Textes (bzw. der eigentlichen Daten):* Bei HTML- oder XML–Dateien müssen die Markup–Tags entfernt werden. Aus PDF-Dateien

[8] vgl. auch Abschnitt 5.2.4.

[9] vgl. hier und im Folgenden z.B. [MRS07], S. 17ff.

oder Texten, die im (Binär–)Format z.B. einer Office–Anwendung abgespeichert wurden, muss der eigentliche Text extrahiert werden. Bei den Mails einer MBOX–Datei können z.B. evtl. die Header–Informationen ignoriert werden.

- *Zeichenkodierung und -Umwandlung:* Auch wenn die immer weiter gehende Durchsetzung von Unicode dieses Problem hoffentlich langfristig entschärfen wird, gibt es immer noch viele unterschiedliche Zeichenkodierungen, welche für Texte bzw. textuell vorliegende Daten benutzt werden. Je nachdem welche Kodierung die Suchanwendung benutzt müssen die Daten evtl. in dieses Format konvertiert werden.[10]

- *Decodierung von Entities:* In HTML- und XML–Dateien sind bestimmte Sonderzeichen (meist aus syntaktischen Gründen) nicht direkt angegeben, sondern als sog. Entities („ü“ statt „ü“, „&“ statt „&“, etc.) kodiert. In LaTeX–Quellcode müssen ebenfalls eine Reihe von Zeichen mittels spezieller Kommandosequenzen eingegeben werden. In solchen Fällen ist ggf. ebenfalls eine Umwandlung erforderlich.

Token–Erzeugung

Nachdem der eigentliche Text der Dokumente vorliegt, müssen die einzelnen, in den Index aufzunehmenden Terme extrahiert werden. Hierzu wird der Text in einzelne Token unterteilt. Wie genau diese Einteilung erfolgt, hängt wiederum von einer Reihe von Faktoren ab und ist mitunter alles andere als eine triviale Aufgabe.

Eine intuitiv eingängige und — insb. bei „normalem Text“ — oft sinnvolle Methode ist die Aufteilung in Terme anhand der Wortzwischenräume („whitespace“) bzw. Satzzeichen.[11] Allerdings gibt es z.B. Sprachen, in denen Texte gänzlich ohne Zwischenräume geschrieben werden (z.B. Chinesisch) und in denen alleine aus diesem Grund dieses Verfahren nicht möglich ist.

Weitere Probleme stellen sich bei der Behandlung z.B. von zusammengesetzten Wörtern — sollen diese besser als ein Term oder aufgeteilt als mehrere Terme in den Index aufgenommen werden? Spezielle Textbestandteile wie z.B. e–Mail–Adressen oder URLs sollten, obwohl sie Sonderzeichen enthalten, normalerweise als ein zusammenhängendes Token erkannt werden. Satzzeichen werden normalerweise bei der Token–Erzeugung entfernt; aber auch sie können manchmal Teil eines zusammenhängenden Begriffes sein.

Man erkennt also leicht, dass die beste Vorgehensweise bei der Token–Erzeugung

[10] vgl. hierzu auch Abschnitt 4.3.2.

[11] Oder evtl. allgemeiner noch an allen Nicht–Buchstaben–Zeichen.

von der Sprache des Textes bzw. allgemeinen der genauen Art der Daten und auch vom Zweck der Suchanwendung abhängt.

Stemming

Mit Stemming bezeichnet man die Reduzierung bzw. Zusammenführung von verwandten Worten auf einen einzigen Stamm. So könnten z.B. die Wörter „schreiben“, „Schreiber“ und „schreibend“ alle auf den Stamm „schreib“ reduziert werden. Wie man sieht muss der entstehende Stamm auch nicht unbedingt selbst ein gültiges Wort sein, es handelt sich vielmehr um eine rein algorithmische Zusammenfassung.

Vorteile bringt Stemming insofern, als dass die Suche nach einem Term auch Dokumente liefert, in denen ein ähnlicher bzw. verwandter Term vorkommt, was in vielen Fällen sinnvoll sein kann.[12] Des weiteren wird die Größe des Wörterbuches verringert, da nicht mehr viele verschieden Formen eines Begriffes, sondern nur noch die Stammform enthalten sind.

Andererseits kann Stemming auch Nachteile mit sich bringen: So wäre es sicherlich sinnvoll, Dokumente in denen ein gesuchter Term in der exakt gleichen Form vorkommt, höher zu bewerten, als solche, in denen nur eine verwandte Form enthalten ist. Sind im Index allerdings nur die Stammformen gespeichert, ist das nicht möglich.

Entfernen von Stopwörtern

Ein Term der in sehr vielen Dokumenten benutzt wird, trägt im Normalfall wahrscheinlich eher wenig spezielle Information in sich und kann kaum dazu dienen, interessante von uninteressanten Dokumenten zu trennen.[13] Aus diesem Grund kann es sinnvoll sein, solche Terme, sog. Stopwörter (engl. „stopwords“) erst garnicht in den Index mit aufzunehmen. Da diese Terme in vielen Dokumenten und u.U. dort an vielen Stellen vorkommen, spart man sich dadurch z.B. Speicherplatz. „a“ oder „the“ sind z.B. typische Stopwörter für englische Texte.

Die Entfernung von Stopwörtern kann allerdings auch Probleme bereiten. Bei der Suche nach Phrasen z.B. kann es problematisch sein, wenn ausgefilterte Stop-

[12] Dies ist allerdings nicht mit der Synonymsuche zu verwechseln. Während dort auch nach Wörtern gesucht wird, die von ihrer *Bedeutung* her ähnlich sind, wird hier auch nach Wörtern mit ähnlicher *Schreibweise* bzw. etymologischer Herkunft gesucht. Insofern liegt hier eher ein spezieller Fall einer Ähnlichkeitssuche vor.

[13] vgl. auch TF/IDF (Term Frequency/Inverse Document Frequency); ein Maß für die „Wichtigkeit“ von Wörtern in einem Dokument bzw. einer Sammlung von Dokumenten. Siehe z.B. http://en.wikipedia.org/wiki/Tf-idf.

wörter Teil der zu suchenden Phrase sind. Hier werden dann evtl. zu viele und nicht exakt passende Treffer zurückgeliefert.

Mittlerweile gibt es andere Ansätze als Stopwörter, um mit oft vorkommenden Wörtern umzugehen und die daraus resultierenden Nachteile bzgl. Speicherplatz und Abfragezeit zu umgehen. Durch Kompression des Index kann der Speicherplatzbedarf erheblich verringert werden, ohne auf die Einbeziehung häufiger Wörter zu verzichten. Durch Sortierung der Postings–Listen nicht einfach nach Dokument–Nummer sondern z.B. nach einer sinnvollen Termgewichtung kann die Bearbeitung der Liste für einen Term relativ früh abgebrochen werden, da Dokumente mit geringer Gewichtung für diesen Term wohl nicht mehr interessant sind.[14]

Erstellen der eigentlichen Datenstruktur

Die eigentliche Erstellung des Index umfasst das Anlegen des Wörterbuches aus den extrahierten Termen und zugehörigen Dokument- bzw. Terminformationen.

4.2 Der Lucene–Index

4.2.1 Beschreibung der Daten- und Dateistruktur

Beim Lucene–Index[15] handelt es sich um einen normalen Invertierten Index, allerdings mit der Besonderheit, dass auch strukturierte Daten unterstützt werden. Im Gegensatz zum normalen Invertierten Index, der in seinem Wörterbuch nur eine Liste von Termen verwaltet, erlaubt das Lucene–Indexformat mehrere unterschiedliche Felder pro Dokument.

Segmente

Ein Lucene–Index besteht aus einem oder mehreren Segmenten (engl. „segments“). Jedes dieser Segmente stellt von seiner Struktur her einen eigenen, selbstständigen Index dar und könnte im Prinzip alleine als Grundlage für eine Suche dienen. Allerdings enthält natürlich nur der Gesamtindex, d.h. alle Segmente zusammen,

[14] vgl. z.B. [MRS07], S. 23ff bzw. S. 107ff.

[15] Die Index–Datei–Formate der Lucene–Implementierungen sind untereinander binärkompatibel (siehe z.B. [ASFLf]), so dass die Beschreibung für alle Lucene–Implementierungen, auch für die in dieser Arbeit verwendete CLucene–Engine, gültig ist; für die folgende Beschreibung des Indexformats vgl. [ASFLiff].

auch die Gesamtheit der Terme, Dokumentverweise und andere Informationen des zugrunde liegenden Datenbestandes.

Die Unterteilung in einzelne Segmente hat nur implementationstechnische Gründe; beim Hinzufügen neuer Dokumente zum Gesamtindex z.B. erweist es sich als einfacher, ein neues, kleines Segment anzulegen anstatt die neuen Informationen direkt in den womöglich bereits sehr großen Gesamtindex einzufügen.

Die einzelnen Segmente bestehen jeweils aus einer Anzahl von Dateien[16] unterschiedlichen Typs. In diesen Dateien werden die Informationen über die einzelnen Felder, die Daten der „stored fields“ (s.u.), Normalisierungsfaktoren, Termvektoren, gelöschte Dokumente und, als wichtigster Bestandteil, das Wörterbuch mit Termliste, Häufigkeits- und Positionsdaten abgespeichert.

Felder

Wie bereits erwähnt, können für jedes Dokument Daten für mehrere Felder abgespeichert werden. Eine weitere Besonderheit hierbei ist, dass Felder als sog. „stored fields“ deklariert werden können. Der Inhalt dieser Felder wird nicht nur indexiert, sondern es wird auch der Originalinhalt des Feldes zusätzlich unverändert im Index abgespeichert.

Es ist sogar möglich, Inhalte nur zu speichern ohne das entsprechende Feld überhaupt zu indexieren. Die entsprechenden Inhalte sind dann zwar nicht durchsuchbar, sind aber direkt verfügbar, wenn das entsprechende Dokument als Treffer einer Suche auftaucht. So ist es möglich, diese Daten z.B. ebenfalls in der Ergebnisliste anzuzeigen, ohne hierzu extra noch einmal auf das extern gespeicherte Dokument zugreifen und die Daten jeweils noch zusätzlich auslesen zu müssen.

Die Informationen über die einzelnen Datenfelder sind in drei Dateien abgespeichert:

- Die *FieldInfos–Datei* (Endung `.fnm`) enthält für alle in den Dokumenten (des aktuellen Segments) vorkommenden Felder den Feldnamen, eine (per Segment) eineindeutige, automatisch vergebene ID–Nummer und eine Handvoll Flags, welche spezielle Eigenschaften des Feldes markieren können.

- Die *FieldData–Datei* (Endung `.fdt`) enthält die eigentlichen abgespeicherten Daten der „stored fields“. Für jedes Vorkommen eines Feldes in einem Dokument wird hier ein Datensatz angelegt. Es ist erlaubt, dass ein Feld

[16] In neueren Lucene–Versionen sind diese Einzeldateien in einer einzigen Archiv–Datei zusammengefasst.

pro Dokument mehrmals vorkommt[17], so dass ein Datensatz wiederum aus mehreren Unter–Datensätzen bestehen kann, deren Anzahl hier angegeben wird. Jeder dieser Unter–Datensätze besteht aus der ID–Nummer des Feldes, einigen Flags für bestimmte Eigenschaften und dem eigentlichen Feldinhalt.

- Um einen schnellen Zugriff auf die Daten der „stored fields“ zu erlauben, wird in der *FieldIndex–Datei* (Endung `.fdx`) eine Zeigerliste verwaltet, in der jeweils die Position der Felddaten der Dokumente vermerkt ist. Da die Größe dieser Zeiger konstant ist, ist so ein Direktzugriff auf die Daten möglich.

Wörterbuch

Die Verwaltung der Terme für separate Felder geschieht beim Lucene–Index über eine Erweiterung der Definition des Termbegriffs. In einem normalen Invertierten Index bestehen die Terme nur aus dem eigentlichen textuellen Inhalt; beim Lucene–Index definiert sich ein Term dagegen zusätzlich über den Namen des Feldes, dem er entnommen wurde, d.h. Terme sind hier Paare aus Feldname und eigentlichem Text.[18] Dieses Verfahren erlaubt es, alle Terme in einem einzigen Wörterbuch abzuspeichern.

Das Wörterbuch besteht aus zwei Dateien:

- Die *TermInfo–Datei* (Endung `.tis`) enthält die eigentlichen Termlisten–Daten. Die Terme werden lexikographisch sortiert; erst nach dem Feldnamen, dann nach dem Term–Text.

  Ein Term wird repräsentiert durch eine Feld–ID–Nummer, eine Präfix–Längenangabe n und einen Suffix–Text. Den vollständigen Term–Text erhält man, indem man die ersten n Zeichen des vorherigen Terms als Präfix nimmt und den aktuellen Suffix–Text anhängt; dieser Text wird dann noch ergänzt durch den Feldnamen, der durch die ID–Nummer spezifiziert wird.

  Neben dem Text enthalten die Term–Datensätze noch eine Angabe, in wievielen Dokumenten dieser Term insgesamt vorkommt und drei Zeiger, welche angeben, wo in den entsprechenden Dateien (s.u.) die Häufigkeits- bzw. Positionsdaten für diesen Term zu finden sind.

[17] Ein Feature das beim DBLP z.B. Für die Verwaltung mehrere Autoren einer Publikation sehr nützlich ist.

[18] Wobei allerdings nicht etwa jeweils der vollständige Feldname im Wörterbuch mit abgespeichert wird; s.u.

- Die *TermInfoIndex–Datei* (Endung `.tii`) dient dazu, den Zugriff auf die Term–Daten zu beschleunigen. Es handelt sich hierbei quasi um eine „ausgedünnte“ Version der TermInfo–Daten: Die Datei enthält jeden *n*–ten Eintrag der TermInfo–Datei, sowie einen Zeiger auf die Position dieses Eintrags in ebendieser Datei. Durch diese Konstruktion wird praktisch ein wahlfreier Zugriff (Direktzugriff) auf bestimmte Positionen innerhalb der großen Termliste ermöglicht.

  Der Wert von *n* ist konfigurierbar und beeinflusst u.A. die Schnelligkeit mit der Terme im Wörterbuch gefunden werden sowie andererseits die Größe der TermInfoIndex–Datei. Große Werte für *n* führen zu einem kleineren Index aber längeren Suchzeiten, kleine Werte führen zu größerem Index aber schnellerem Finden von Termen.

Term–Vektoren

Lucene stützt sich zwar hauptsächlich auf die Datenstruktur des Invertierten Index, erlaubt aber auch die Erzeugung und Speicherung von Term–Vektoren (auch: Dokument–Vektoren). Diese werden im Vektorraum–Modell[19] verwendet, um die erfassten Dokumente als Vektoren darzustellen.

In einer sehr einfachen Version des Vektorraum–Modells (engl. „vector space model“) wird jedem Term, der in der Dokumentsammlung vorkommen kann, eine Dimension eines Vektorraumes zugeordnet. Ein Dokument kann dann als Punkt (Vektor) innerhalb dieses Vektorraumes aufgefasst werden, indem man z.B. die Häufigkeiten der Vorkommen der einzelnen Terme in diesem Dokument zählt und den entsprechenden Wert jeweils an der entsprechenden Stelle für den Term im Dokument–Vektor einträgt.[20]

Suchabfragen können in ebensolcher Weise dargestellt werden. Die Auswertung erfolgt dann, indem man Dokument–Vektoren sucht, die möglichst ähnlich zum Abfrage–Vektor sind.

Da allerdings diese gespeicherten Term–Vektoren in der im Zuge dieser Arbeit entwickelten Applikation keine direkte Anwendung finden, wird auf die genaue, im Lucene–Index zur Speicherung benutzte Daten- bzw. Dateistruktur hier nicht weiter eingegangen.

[19] vgl. z.B. [WPvsm].
[20] vgl. auch mit der Inzidenz–Matrix (→ 4.1.1)

Normalisierungs–Faktoren

Diese Dateien enthalten für jedes Feld in jedem Dokument einen Wert, der bei der Berechnung der Score (→ 5.2.4) berücksichtigt wird, wenn dieses Dokument bzw. Feld in der Ergebnismenge einer Abfrage auftaucht.

Deletions

Die *Deletions–Datei* (Endung `.del` verzeichnet gelöschte Dokumente innerhalb eines Segments. Dokumente können bei dynamisch aktualisierten Indizes auftreten. Wird der Index bei jeder Änderung oder auch periodisch komplett neu erstellt ist diese Datei nicht vorhanden.

Ähnlich wie neue Daten aus Performance–Gründen nicht direkt in eine große Hauptindex–Datei eingepflegt werden, werden auch beim Löschen von Dokumenten deren Daten nicht direkt aus den Segment–Dateien entfernt. Stattdessen wird in dieser Datei eine Liste der gelöschten Dokumente verwaltet. Die Listen werden als Bitvektoren mit einem Bit pro indexiertem Dokument oder als DGaps[21] repräsentiert. Gesetzte Bits entsprechen dabei gelöschten Dokumenten.

4.2.2 Die bei der Generierung relevanten Klassen

Die für den Nutzer der Lucene–API[22] interessanten Klassen zur Erstellung eines Index sind in ihrer Anzahl recht begrenzt und einfach anzuwenden.

Die wichtigste Klasse ist der `IndexWriter`, der für die eigentlich Konstruktion und Wartung bzw. Aktualisierung des Index zuständig ist. Für die Analyse und Verarbeitung der Daten stützt sie sich auf eine (Unter–)Klasse vom Typ `Analyzer`, die wiederum mit Hilfe von `Tokenizern` und `TokenFiltern` die einzelnen Schritte zur Erzeugung der Terme ausführt.

Aufbereiten der Dokumente

Da es sich bei Lucene lediglich um eine allgemein Bibliothek und nicht um eine spezielle Suchanwendung handelt, stellt Lucene keine eigenen Klassen bzw. Code

[21] Eine Methode zur Komprimierung von Bitvektoren, die insb. bei sparsam besetzten Vektoren, wie sie hier vorliegen, Speicherplatz spart; siehe http://bmagic.sourceforge.net/dGap.html.

[22] Wie beim Index–Format so wurde — zumindest beim CLucene–Port — auch bei der Struktur und den Schnittstellen der Klassen darauf geachtet, diese so eng wie möglich an die Java–API anzulehnen, so dass diese Beschreibung wiederum sowohl für die Java- als auch C++–Engine gilt. Zur Beschreibung der Klassen und deren Anwendung vgl. hier und im Folgenden u.A. die entsprechenden Seiten in [ASFLjd]

für das Aufbereiten der Dokumente bereit.

Die mitgelieferte Demo–Anwendung kann allerdings als ein erstes Beispiel dienen, wie z.B. eine Sammlung von Dateien für die Suche aufbereitet wird. Daneben gibt es mehrere weitere Projekte, die auf Lucene aufsetzen und die in dieser Hinsicht weiterhelfen können. Zu nennen sind hier z.B. Nutch, eine Websuchen–Software[23], die Parser für HTML und andere Dateiformate bereitstellt und Solr, ein allgemeiner Suchserver [24].

Token–Erzeugung

Die Token–Erzeugung ist ein Teil der Aufbereitung, der von der dem `IndexWriter` zugeordneten `Analyzer`–Klasse übernommen wird. Diese greift dafür wiederum auf eine spezielle `Tokenizer`–Klasse zurück.

Von Lucene bereitgestellt werden z.Zt. vier Tokenizer:

LetterTokenizer Diese Klasse bildet Token, indem sie den Zeichenstrom an Nicht–Buchstaben–Zeichen unterteilt.

WhitespaceTokenizer Diese Klasse unterteilt den Zeichenstrom an Whitespace–Vorkommen (Leerzeichen, Tabulatoren, etc.)

KeywordTokenizer Hierbei handelt es sich um einen recht einfachen Tokenizer, der keinerlei Unterteilung vornimmt und einfach die gesamte Zeichenkette ohne Veränderung als ein Token zurückliefert.

StandardTokenizer Dieser Tokenizer eignet sich (laut Lucene–API–Dokumentation) gut für die meisten Europäischen Sprachen. Token werden erzeugt, indem der Zeichenstrom an Satzeichen und Whitespace aufgetrennt wird, wobei diese entfernt werden. Punkte werden dann als Teil eines Token aufgefasst, wenn sie *nicht* von einem Whitespace–Zeichen gefolgt werden, wodurch z.B. Java–Paketnamen wie „org.apache.lucene“ als ein Token behandelt werden. Durch Bindestriche verbundene Worte werden aufgeteilt, außer wenn sie eine Nummer enthalten. e-Mail-Adressen und Internet–Domainnamen werden ebenfalls erkannt und als ein Token behandelt.

Für die Java–Lucene–Version existieren darüber hinaus noch eine Reihe weiterer (allerdings zum Teil nicht „offizieller“) Tokenizer wie z.B. `ChineseTokenizer` (speziell für Chinesische Texte), `NGramTokenizer` und `EdgeNGramTokenizer`

[23] siehe http://lucene.apache.org/nutch/about.html
[24] siehe http://lucene.apache.org/solr/

(zum Zerlegen des Textes in N–Grams[25]) und eine Reihe von Tokenizern für weitere Sprachen wie z.B. Russisch.

Teilweise stellen diese Tokenizer allerdings bereits eine Kombinationen von Tokenizer und Filter dar: Der `LowerCaseTokenizer`, welcher neben der Aufteilung des Textes in Token wie beim `LetterTokenizer` auch bereits die Umwandlung dieser Token in Kleinschreibung durchführt, ist so ein Beispiel.

Stemming

Stemming wird ebenfalls unter dem Dach einer Analyzer–Klasse, mit Hilfe eines Filters, durchgeführt. In der CLucene–Bibliothek, allerdings nicht im „core“ sondern nur im „contrib“–Paket, enthalten sind:

PorterStemmer Dieser Stemmer benutzt zur Behandlung der Token den sehr weit verbreiteten Porter–Stemming–Algorithmus[26].

Snowball–Klassen Hierbei handelt es sich um Snowball[27]–generierte Stemmer–Klassen für verschiedene Sprachen.

Entfernen von Stopwörtern

Zum Entfernen von Stopwörtern stehen zwei Filter zur Verfügung:

StopFilter Dieser Filter entfernt Token anhand einer von der Anwendung vorgegebenen Liste von Stopwörtern.

LengthFilter Mit Hilfe dieser Klasse können zu kurze oder zu lange Token ausgefiltert werden.

Sonstige Filter

Der `ISOLatin1AccentFilter` wandelt akzentuierte Zeichen aus dem „ISO Latin 1“–Zeichenvorrat (ISO-8859-1) in die einfache Form, ohne Akzent, um.

Der `LowerCaseFilter` wandelt alle Terme in Kleinschreibung um. Dieser Filter ist einer der vom `StandardAnalyzer` benutzten Filter.

[25] siehe z.B. http://en.wikipedia.org/wiki/Ngram

[26] siehe http://tartarus.org/~martin/PorterStemmer/index.html

[27] Eine Sprache zur einfachen Repräsentation von Stemming–Algorithmen; siehe http://snowball.tartarus.org/

Erstellen der eigentlichen Datenstruktur

Das Erstellen der Datenstrukturen sowie das Schreiben und die Verwaltung der beteiligten Daten wird von der oben erwähnten Klasse `IndexWriter` übernommen.

Die eigentliche Anwendung generiert für jedes zu indexierende Dokument Objekte vom Typ `Document`, denen dann die entsprechenden `Fields` hinzugefügt werden. Diese Dokumente werden dann wiederum dem Index(Writer) hinzugefügt.

4.3 Das Programm `dblp2index`

Speziell zum Zweck der Konvertierung der DBLP–XML–Daten aus der Datei `dblp.xml` in einen CLucene–Index wurde das Programm `dblp2index` geschrieben. Es parst die XML–Datei, erkennt die einzelnen Publikationen–Datensätze und Datenfelder, generiert daraus die entsprechenden `Document`- und `Field`–Objekte und fügt diese dem Index hinzu.

4.3.1 Bedienung

Aufruf

Der Aufruf des Programms erfolgt über die Kommandozeile. Das Programm akzeptiert vier Parameter, deren Angabe jedoch nur nötig wird, falls die eingebauten Standardwerte nicht passend sein sollten. Die Parameter im Einzelnen:

- `-i` („Input“): Durch Angabe von `-i <dateiname>` kann bestimmt werden, aus welcher DBLP–XML–Datei die Daten gelesen werden sollen. Als Standardwert vorgegeben ist „dblp.xml“, d.h. die Eingabedaten werden aus der Datei `dblp.xml` im aktuellen Verzeichnis gelesen.

- `-o` („Output“): Durch Angabe von `-o <verzeichnisname>` kann bestimmt werden, wie das Verzeichnis, in dem die Daten des zu erstellenden Index erzeugt werden, heißen soll. Das Verzeichnis wird automatisch angelegt, sollte es noch nicht existieren. Als Standardwert vorgegeben ist „index“, d.h. der Index wird in einem Unterverzeichnis `index` des aktuellen Verzeichnisses abgelegt.

- `-q` („Quiet“): Durch Angabe von `-q` können die Ausgaben des Programms eingeschränkt werden. Normalerweise wird während des Indexierens für alle 1000 indexierten Publikationen eine Fortschrittsmeldung ausgegeben. Ist dieser Parameter angegeben, werden diese Meldungen nicht angezeigt.

- `-?`: Der Parameter `-?` liefert eine Syntaxhilfe und ein paar kurze Informationen zum Programm.

Wichtig: Die Datei `dblp.dtd` muss sich im aktuellen Verzeichnis befinden, da sie von `dblp.xml` in dieser Weise referenziert wird!

Ausgaben und erstellte Dateien

Sollte der Parameter `-q` nicht angegeben worden sein, wird während des Indexierens alle 1000 Publikationen eine Fortschrittsmeldung ausgegeben. Außerdem wird in diesem Fall die für die Erstellung des Index benötigte Zeit gemessen und nach Fertigstellung ebenfalls angezeigt.

Sollte es noch nicht existieren wird ein Verzeichnis angelegt, welches die Datendateien des neu erstellten Index aufnimmt (s.o., Parameter `-o`). Sollte ein entsprechendes Verzeichnis bereits existieren *werden alle evtl. bereits darin befindlichen Dateien gelöscht* und ein neuer Index wird erstellt.

Nachdem der Index vollständig ist, werden zum Abschluss statistische Informationen zur Anzahl der indexierten Publikationen, der Anzahl der erfassten Einzelterme und die interne CLucene–Versionsnummer des Index ausgegeben.

Fehlerbehandlung und Rückgabecodes

Während der Indexerstellung auftretende Fehler (Exceptions) werden gefangen und eine entsprechende Meldung wird ausgegeben. In einem solchen Fehlerfall ist der Rückgabewert des Programms 1. Im Normalfall, d.h. bei erfolgreichem Durchlauf, wird 0 zurückgegeben.

4.3.2 Programmablauf

Das Programm besteht im Wesentlichen aus zwei Teilen: Dem Hauptprogramm `dblp2index` und der Klasse `CLuceneIndexBuilder`.

Die Aufgaben der Hauptprogramms beschränken sich auf das Auswerten eventueller Kommandozeilen–Argument zur Konfiguration, dem Aufruf der Methode zum Bau des Index und der Ausgabe einiger statistischer Daten nach erfolgreichem Durchlauf.

Die eigentliche Arbeit wird von der Klasse `CLuceneIndexBuilder` ausgeführt. Diese Klasse leitet sich von `DBLPParser` (siehe Anhang) ab und definiert bzw. überschreibt die zwei Methoden `buildIndex` und `publication`.

Methode `buildIndex`

Diese Methode erhält als Parameter den Namen der einzulesenden XML–Datei, den Namen des zu erstellenden Index sowie ein Flag, das die Fortschrittsanzeige steuert.

Die Methode öffnet und konfiguriert den `IndexWriter`; dann stößt sie das Parsen der Datei mittels der von ihrer Basisklasse übernommenen `parse`–Methode an, die im Zuge dessen für jeden gefundenen Publikationen–Datensatz wiederum die `publication`–Methode (s.u.) aufruft, in welcher die Hauptarbeit für die Indexerstellung stattfindet.

Nachdem das Parsen und die Indexerstellung abgeschlossen sind, wird dieser noch optimiert, d.h. die entstandenen Einzelsegmente (→ 4.2.1) werden geordnet und zu einer großen Datei zusammengefasst. Schließlich wird der `IndexWriter` geschlossen und zum Hauptprogramm zurückgekehrt.

Methode `publication`

Diese ebenfalls von der Basisklasse ererbte und hier überschriebene Methode führt die eigentlich interessante Arbeit aus.

Die Methode erhält ein Objekt vom Typ `dblp::Publication` übergeben, das bereits die ausgewerteten Informationen zum Publikationen–Datensatz und den enthaltenen Feldern in strukturierter und leicht zugreifbarer Form bereitstellt.

Ausgehend von der Syntax bzw. Semantik der in den einzelnen Datenfeldern enthaltenen Informationen werden entsprechende `Field`–Objekte erzeugt und konfiguriert.

Diese Objekte werden dann dem ebenfalls hier erzeugten CLucene–`Document`–Objekt angehängt; dieses wiederum wird am Schluss in den Index eingefügt.

Indexierte Felder

Da CLucene die Möglichkeit bietet, pro Dokument praktisch beliebig viele Felder zu verwalten, könnten im Prinzip auch alle im DBLP vorkommenden Felder berücksichtigt werden. Andererseits macht es keinen Sinn, Felder, deren Daten mit sehr großer Wahrscheinlichkeit für den „normalen Nutzer“ uninteressant sind, wirklich mit in den Index aufzunehmen.

Aus diesem Grund werden die beiden DBLP–Felder „url“ und „cdrom“ vollständig ignoriert, d.h. weder indexiert noch direkt im Index gespeichert, da beide Felder nur für interne Zwecke der DBLP–Website bzw. der ACM SIGMOD An-

thology verwendet werden.[28]

Sollten sich später doch sinnvolle allgemeine Anwendungmöglichkeiten auch für diese Daten ergeben, so ist es ohne Probleme möglich, die entsprechenden Felder ebenfalls im Index aufzunehmen.[29]

Das Feld „ee“ (Electronic Edition) wird ebenfalls nicht indexiert, da es wohl sehr unwahrscheinlich ist, dass ein Nutzer Publikationen anhand einer (bereits bekannten!) URL einer digitalen Ausgabe dieser Publikation suchen wird. Da der Inhalt dieses Feldes allerdings für die Anzeige in der Ergebnisliste der neuen DBLP–Suche benötigt wird, werden die entsprechenden Daten gespeichert.

Um dem Nutzer dennoch eine einfache Möglichkeit zu bieten, nach Publikationen zu suchen, für die Informationen über eine EE vorliegen, wird stattdessen ein Feld „hasEE“ in den Index aufgenommen. Dieses erhält den Wert „1“ wenn ein Eintrag „ee“ vorhanden ist und „0“ sonst.

Für alle anderen DBLP–Datenfelder besteht zumindest eine gewisse Wahrscheinlichkeit, dass sie bei einer Suche angefragt werden könnten, weshalb diese Felder auch in den Index übernommen werden. Die Aufteilung in CLucene–`Fields` folgt hierbei weitestgehend der Aufteilung der Datenfelder im DBLP, d.h. die Datenfeldelemente in `dblp.xml` werden als CLucene–`Field`–Objekt, mit dem Element–Tag/Namen als Feldnamen, in den Index eingefügt.

Zusätzlich zu diesen, direkt durch XML–Elemente repräsentierten DBLP–Datenfeldern werden auch der Publikationstyp (→ 2.2.1), der DBLP–Publikations–Key und das Änderungsdatum (→ 2.2.2) als Felder mit Namen „pubType“ bzw. „key“ und „mdate“ in den Index aufgenommen.

Einerseits besteht auch hier zumindest eine gewisse Wahrscheinlichkeit für ein Nutzerinteresse an diesen Daten; sei es, wie bereits erwähnt, z.B. eine Suche nach kürzlich geänderten Publikationsdatensätzen mittels „mdate“ oder evtl. eine aus externer Quelle erhaltene Referenz über einen DBLP–„key“, zu dem nun die entsprechende Publikation gesucht werden soll. Außerdem wird der DBLP–Schlüssel auch für die Anzeige der Ergebnisliste benötigt, um die Verweise auf die BibTEX–Seiten des DBLP zu konstruieren.

[28] „url“ enthält die URL des Table Of Contents der Eltern–Publikation für Publikationen, die in einer Sammlung enthalten sind; „cdrom“ wird nur innerhalb der ACM SIGMOD Anthology benutzt; vgl. [LR06], S. 125 bzw. S. 127.

[29] Außer der Modifikation einer einzigen `if`–Abfrage im Programm `dblp2index` wären dafür keinerlei weitere Maßnahmen erforderlich.

Gespeicherte Felder

Zwar ist dieser Punkt bei den heutigen Festplattengrößen eher von ungeordneter Bedeutung; um Speicherplatz zu sparen werden allerdings trotzdem die meisten der Felder nicht als „stored fields“ deklariert; lediglich für „author“, „title“, „year“, „ee“ und „key“ werden in dieser Weise die Feldinhalte auch zusätzlich direkt im Index gespeichert, da diese Werte auch bei der Anzeige der Ergebnisliste benötigt werden.

Auf die Geschwindigkeit der Suchabfragen sollte die Speicherung oder Nichtspeicherung der Feldinhalte übrigens keine Auswirkungen haben, da die „stored fields“ separat vom Index–Wörterbuch verwaltet werden (→ 4.2.1). Genauere Testergebnisse hierzu sind allerdings nicht verfügbar.

Token–Erzeugung

Der Inhalt der meisten Felder wird „tokenized“, d.h. der Analyse und Aufteilung in einzelne Terme unterworfen. Hierzu wird der `StandardAnalyzer` verwendet. Dieser benutzt zur eigentlichen Zerlegung den `StandardTokenizer` (→ 4.2.2), dessen Vorgehensweise für die Inhalte der DBLP–Datenfelder im Normalfall sinnvolle Ergebnisse liefern sollte. Felder wie z.B. „title“ oder „booktitle“ enthalten in der Regel deutschen oder englischen Text für dessen Behandlung diese Klasse speziell ausgelegt ist.

Bei anderen DBLP–Feldern ist dagegen eine derartige Behandlung entweder unnötig oder könnte sogar zu unsinnigen Ergebnissen führen. Die Felder „chapter“, „hasEE“, „month“, „volume“ und „year“ enthalten ganz normale Zahlen, die bereits ein vollständiges Token darstellen und nicht weiter aufgetrennt werden sollten. Der Inhalt von „pubType“ besteht ebenfalls bereits nur aus jeweils einem einzigen Wort. „cite“, „crossref“ und „key“ enthalten jeweils einen DBLP–Schlüssel, der an sich eine vollständige Einheit darstellt, ähnlich wie die „isbn“ und „mdate“.

Groß-/Kleinschreibung

Für die mittels des Analyzers behandelten Felder werden die entstandenen Token durch den `LowerCaseFilter` in durchgängige Kleinschreibung überführt. Zwar wird hierdurch eine Suche mit exakter Übereinstimmung von Groß-/Kleinschreibung des Suchterms mit den Termen im Index verhindert; allerdings dürfte in praktisch allen Fällen die exakte Schreibweise hier kein relevantes Unterscheidungskriterium darstellen, durch welches wirklich zwei signifikant unterschiedliche Begriffe voneinander getrennt werden.

Wahrscheinlich ist eher sogar davon auszugehen, dass bei einer Berücksichti-

gung der Schreibweise die Wahrscheinlichkeit groß ist, dass der Nutzer dieses zusätzliche Kriterium nicht realisiert und auf diese Weise eigentlich relevante Treffer unter den Tisch fallen. Aus diesem Grunde, und auch um die Größe des Index nicht durch die entstehenden zusätzlichen Terme zu erhöhen, wurde die Umwandlung der Terme in Kleinschreibung beibehalten.[30]

Stemming

Der `StandardAnalyzer` benutzt außerdem die `StandardFilter`–Klasse; diese entfernt „'s“ von Wörtern (was meist bei englischsprachigen Texten/Daten vorkommen dürfte, von denen jedoch im DBLP auch eine große Anzahl enthalten sind, insb. durch die vielen englischsprachigen Publikationstitel). Außerdem werden die Punkte von Akronymen entfernt.

Wirkliches Stemming oder eine andere Vereinheitlichung von Wortformen wird nicht durchgeführt. Da eine Suche nach verschiedenen Schreibweisen eines Begriffes mittels der Ähnlichkeits- bzw. Wildcard–Suche ohne weiteres möglich ist, sollten hierdurch allerdings keine Nachteile entstehen.

Entfernen von Stopwörtern

Der weiterhin vom `StandardAnalyzer` angewendete `StopFilter` entfernt eine kleine Menge von Wörtern anhand einer vorgegebenen, fest einkodierten Wortliste. Diese umfasst 35 sehr häufige englische Wörter wie z.B. „a“, „and“, „by“ oder „the“. Darin enthalten sind auch die „Wörter“ „s“ und „t“, die aufgrund der Art der Token–Erzeugung (Aufteilen an und Entfernen von speziellen Zeichen) aus Begriffen wie „don't“ oder „it's“ übrigbleiben (s.o.; wobei „s“ allerdings bereits durch den `StandardFilter` entfernt wurde).

Durch diese Maßnahme wird sowohl die Größe des Index als auch die Zeit für die Auswertung von Suchabfragen verringert. Bei der Phrasensuche macht der Verzicht auf die Aufnahme dieser Wörter keine Probleme (→ 4.1.3), da dort im Normalfall eine Art Ähnlichkeitssuche angewendet wird (→ 5.2.1) durch die auch Phrasen, in denen diese Wörter vorkommen bzw. fehlen, gefunden werden können.

[30] Sollte sich im Laufe der Zeit herausstellen, dass eine Berücksichtigung von Groß-/Kleinschreibung doch sinnvoll oder gewünscht ist, kann dies durch Implementierung einer eigenen Analyzer–Klasse ohne großen Aufwand realisiert werden.

Sonderzeichen und Zeichenkodierung im Index

Ein nicht zu vernachlässigender Punkt ist, welche Zeichenkodierung für den Index — und damit auch für die Suche — verwendet wird.

Sonderzeichen sind in der Datei `dblp.xml`, wie in XML üblich[31], als Entities („ü“, „ß“, etc.) codiert. Beim Auslesen durch den SAX–Parser werden diese Entities, u.A. mit Hilfe der zur Datei gehörigen DTD, in die entsprechenden „normalen“ Zeichen umgewandelt.

Zum Parsen der XML–Datei wird das Arabica–Toolkit[32] benutzt, welches wiederum auf dem Expat–Parser[33] aufsetzt. Die String–Inhalte von Elementen und Attributen werden von Arabica in der UTF–8–Zeichenkodierung zurückgeliefert.[34]

CLucene kann sowohl für die Benutzung von Ein–Byte–Kodierung („ASCII“) als auch Mehr–Byte–Kodierung („UCS2“) konfiguriert werden. In letzterem Fall wird der `wchar_t`–Datentyp benutzt, bei dem es sich im Normalfall um ein 32–Bit–Format handelt.[35] In diesem Fall kann auch der gesamte Umfang des Unicode–Zeichensatzes benutzt werden.

Da in jedem Fall eine Konvertierung des vom XML–Parser gelieferten UTF–8–kodierten Zeichenstroms in das von CLucene benutzte Format nötig ist, sind im Bezug auf die Programmierung beide Varianten gleich gut geeignet bzw. mit mehr oder weniger gleich hohem Aufwand zu benutzen. UTF–8 bietet jedoch wesentlich weitergehende Möglichkeiten und ist insgesamt als die zukunftsweisendere der beiden Alternativen anzusehen, so dass dieser Option letztendlich der Vorzug gegeben wurde.

Die Behandlung der diversen String–Kodierungen wird u.A. mithilfe der neu entwickelten Klasse `TCharString` aus der Bibliothek `libdblp` (siehe Anhang) realisiert, die eine Methode zur Konvertierung (mittels der CLucene–Funktionen `lucene_utf8towcs` bzw. `STRCPY_AtoT`) von Ein–Byte–kodierten Zeichenketten in das CLucene–TCHAR–Format bereitstellt.

Insgesamt ist noch anzumerken, dass der gesamte Bereich der Zeichenkodierung recht komplex ist und zum Teil einige Überlegungen und Arbeit erfordert. Obwohl der vorhandene Code getestet und für voll funktionsfähig befunden wurde, ist leider keinesfalls auszuschließen, dass hier noch Fehler oder Probleme in

[31] vgl. [BPS+06], Abschnitt „4.3.3 Character Encoding in Entities“

[32] siehe http://www.jezuk.co.uk/cgi-bin/view/arabica

[33] siehe http://expat.sourceforge.net/

[34] Da Arabica ein Frontend darstellt und als Backend nicht nur auf Expat, sondern auch auf Xerces oder libxml2 zurückgreifen kann, wäre es evtl. interessant zu sehen, ob die Benutzung eines dieser Parser ein unterschiedliches Verhalten aufweist.

[35] siehe z.B. http://www.cl.cam.ac.uk/~mgk25/unicode.html#c

der Architektur oder Implementierung existieren.

4.3.3 Performance

Zum Vergleich der Performance und des Ressourcenverbrauchs, sowohl bei der Erstellung als auch bzgl. des fertigen Index, wurden einige Tests mit verschiedene Konfigurationen durchgeführt. Die Tests unterscheiden sich hierbei in der Art, welche Felder insgesamt berücksichtigt wurden, und ob diese gespeichert oder nur indexiert wurden. Außerdem wurde der Aufwand für die Token–Erzeugung d.h. Behandlung mit dem Analyzer und den entsprechenden Filtern untersucht.

Der Speicherverbrauch wurde durch manuelle Überwachung während des Programmablaufs mittels des Linux–Programms `top` ermittelt. Diese Methode ist zwar etwas umständlich und nicht übermäßig genau; da aber leider kein einfaches automatisches Verfahren verfügbar war und außerdem die Werte während der gesamten Laufzeit nur sehr unwesentlich schwankten, sollte diese Methode ausreichen.

Angegeben ist der Gesamtspeicherverbrauch (Residual und Shared) während der normalen Erstellungsarbeit, während des periodischen Zusammenfassens der erstellten Einzelsegmente[36] und bei der abschließenden Optimierung des Gesamtindex.

Die unter Gesamtdauer angegebenen Zeiten umfassen die gesamte zur Erstellung des Index benötigte Zeit, inkl. der abschließenden Optimierung. Angegeben ist die Spannweite (Minimum – Maximum) der in jeweils drei Testläufen maschinell vom Programm selbst ermittelte und ausgegebene Dauern.

Testergebnisse

Im ersten Test (siehe Tabelle 4.2) wurden nur solche Felder, die bei der Ausgabe der Treffer benötigt werden, als „stored“ im Index gespeichert. Des weiteren wurden einige uninteressante Felder nicht „indexed“, d.h. sind (auch) nicht durchsuchbar.

Im zweiten Test (siehe Tabelle 4.3) wurden wieder nur die für die Ausgabe benötigten Felder gespeichert; indexiert wurden jetzt aber alle Felder. Die Indexgröße steigt dabei um ca. 14%, die benötigte Zeit um ca. 11%. Auffällig ist auch die stark (46%, 758.784 neue Terme) gestiegene Termanzahl; insb. die Felder „cdrom“, „isbn“ und „url“, die praktisch für jede Publikation unterschiedliche Werte enthalten, dürften massgeblich zu diesem Anstieg beigetragen haben.

[36] siehe z.B. [ASFLjd], Klasse `org.apache.lucene.index.IndexWriter`, Stichwort `mergeFactor`

Tabelle 4.2: Ergebnisse des `dblp2index`–Performancetest 1

Felder stored	author, title, ee, year, key
... nicht indexed	address, cdrom, chapter, isbn, month, note, school, url
... nicht tokenized	pubType, key, mdate
Speicherbedarf normal	5 MiB
... während Aufräumen	12 MiB
... bei Optimierung	bis max. 21 MiB
Gesamtdauer	26,41–27,49 Minuten
Indexgröße	274,6 MiB (287.930.199 Bytes)
...	32% < `dblp.xml`
Term Count	1.647.697 Terme

Tabelle 4.3: Ergebnisse des `dblp2index`–Performancetest 2: Indexierung

Felder stored	author, title, ee, year, key
... nicht indexed	(keine)
... nicht tokenized	pubType, key, mdate
Speicherbedarf normal	5 MiB
... während Aufräumen	12 MiB
... bei Optimierung	bis max. 21 MiB
Gesamtdauer	29,52–30,75 Minuten
Indexgröße	311,5 MiB (326.597.076 Bytes)
...	16% < `dblp.xml`
Term Count	2.406.481 Terme

Im dritten Test (siehe Tabelle 4.4) wurden alle Felder sowohl indexiert als auch gespeichert; die Indexgröße steigt dabei wieder um ca. 29%, die benötigte Zeit um ca. 7% gegenüber dem zweiten Test; die Anzahl der Terme bleibt natürlich gleich.

Tabelle 4.4: Ergebnisse des `dblp2index`–Performancetest 3: Speicherung

Felder stored	(alle)
... nicht indexed	(keine)
... nicht tokenized	pubType, key, mdate
Speicherbedarf normal	5 MiB
... während Aufräumen	12 MiB
... bei Optimierung	bis max. 24 MiB
Gesamtdauer	31,98–33,01 Minuten
Indexgröße	402,5 MiB (422.052.296 Bytes)
...	10% > `dblp.xml`
Term Count	2.406.481 Terme

In den vorherigen Tests wurden bis auf „pubType“, „key“ und „mdate“ alle Felder durch den `StandardAnalyzer` behandelt. Zur Untersuchung der Auswirkungen der Token–Erzeugung auf Zeitaufwand und Ressourcenverbrauch wurden im vierten Testlauf (siehe Tabelle 4.5) eine Reihe von Feldern als „untokenized“ markiert. Wie man sieht hat das insb. auf die Indexgröße positive Auswirkungen; im Vergleich zum vorherigen Fall sinkt sie um 8%. Die Termanzahl dagegen steigt um fast 14%.

Tabelle 4.5: Ergebnisse des `dblp2index`–Performancetest 4: Token–Erzeugung

Felder stored	(alle)
... nicht indexed	(keine)
... nicht tokenized	pubType, key, mdate, chapter, cdrom, cite, crossref, ee, isbn, month, url, volume, year
Speicherbedarf normal	5 MiB
... während Aufräumen	12 MiB
... bei Optimierung	bis max. 25 MiB
Gesamtdauer	28,38–31,88 Minuten
Indexgröße	370,5 MiB (388.447.613 Bytes)
...	2% < `dblp.xml`
Term Count	2.740.787 Terme

Der letzte Testfall (siehe Tabelle 4.6) repräsentiert schließlich die weiter oben erwähnte Konfiguration, welche schlussendlich für die Erstellung des Suchindex benutzt wurde. Die im Vergleich mit dem ersten, „sparsamsten“ (im Sinne das nur wenige Felder gespeichert und einige nicht indexiert wurden) Test nochmals leicht gesunkene Laufzeit ist wohl zum Teil auf die geringere Arbeit bei der Token–Erzeugung zurückzuführen. Wahrscheinlich spielt hier aber vor allem die ebenfalls dadurch verringerte Termanzahl eine Rolle, die weniger Arbeit beim Aufbau des Wörterbuches bedingt.

Tabelle 4.6: Ergebnisse des `dblp2index`–Performancetest 5: Endgültig benutzte Konfiguration

Felder stored	author, title, ee, year, key
. . . nicht indexed	cdrom, ee, url
. . . nicht tokenized	pubType, key, mdate, chapter, cite, crossref, isbn, month, pages, volume, year
Speicherbedarf normal	5 MiB
. . . während Aufräumen	12 MiB
. . . bei Optimierung	bis max. 23 MiB
Gesamtdauer	24,80–25,13 Minuten
Indexgröße	265,5 MiB (278.377.219 Bytes)
. . .	37% < `dblp.xml`
Term Count	1.349.430 Terme

Bewertung

Allgemein lässt sich feststellen, dass die verschiedenen Konfigurationen kaum bis keinen Einfluss auf den während des Programmlaufs benötigten Speicher haben.

Bemerkenswerter sind hier schon die Veränderungen bei der zum Erstellen des Index benötigten Gesamtzeit, die hier von der kürzesten bis zur längsten Dauer um ca. ein Viertel schwanken. In Anbetracht der Tatsache, dass es dabei allerdings um lediglich 8 Minuten geht, ist auch dieser Wert an sich zwar signifikant, aber in der Praxis — bei der wohl von höchstens einem Programmlauf am Tag auszugehen ist — eher wenig relevant.

Auch die Unterschiede in der Indexgröße von ca. 51% von der kleinsten bis zur größten Datei sind insofern nur ein nebensächliches Entscheidungskriterium, als dass, zumindest rein vom Speicherplatz her, die entsprechenden 137 MiB bei der Größe der heutigen Festplatten ebenfalls praktisch zu vernachlässigen sind.

Bzgl. der Dauer von Suchabfragen sollte sich diese Größenänderung auch nicht in erheblichem Maß auswirken; durch die Verwendung einer speziellen Referenzdatei, die einen beschleunigten Zugriff auf die Indexdaten erlaubt (→ 4.2.1) steigt der Aufwand für das Durchsuchen des Wörterbuches nicht linear mit der Gesamtgröße des Index, sondern nur um einen geringeren Faktor. Allerdings stehen auch hier genaue Tests aus.

Auffällig ist übrigens in allen Fällen, dass die entstehende Index–Datei von der Größe her zumindest grob mit der Original–XML–Datei zu vergleichen ist; die z.B. in [ASFLf] angegebene Kompression auf 25–30% des Umfanges der Originaldaten wird bei weitem nicht erreicht.

Bedenkt man allerdings, dass die Daten des DBLP bzw. die vorkommenden Terme wesentlich weniger Redundanz aufweisen als z.B. eine Sammlung von „normalen“ Texten ist dieses Ergebnis weniger verwunderlich. Zwar gibt es bzgl. der Autorennamen, der Jahreszahlen oder auch bei Feldern wie „journal“ oder „booktitle“ eine gewisse Wiederholung; aber insb. Felder wie „isbn“, „url“ oder „key“ sind praktisch für jeden Publikationsdatensatz unterschiedlich und werden auch als unterschiedliche Terme indexiert.

Auch beim Unterschied in der Termanzahl — von der niedrigsten bis zur höchsten beträgt er fast 50% (entspricht 1.391.357 Termen) — ist die Auswirkung dieser Felder deutlich auszumachen. Die Aufnahme dieser Felder mit sehr großem „Wertebereich“ bedingt insb. auch hier einen erheblichen Anstieg der Zahl der Einträge ins Index–Wörterbuch.

4.3.4 Todos und Verbesserungsmöglichkeiten

Wirkliche Probleme gibt es mit dem Programm bzw. der Indexerstellung an sich nicht mehr. Nichtsdestotrotz sind durchaus noch Verbesserungs- bzw. Erweiterungsmöglichkeiten denkbar:

- *Index–Aktualisierung:* Zwar geht die Erstellung eines komplett neuen Index mit um die 30 Minuten (auf dem Testsystem) verhältnismäßig schnell vonstatten. Trotzdem ist davon auszugehen, dass eine Aktualisierung des Index, insb. aufgrund der auch in Zukunft kontinuierlich sich vergrößernden Datenmenge, eine nochmalige wesentliche Verkürzung des Zeitaufwandes ermöglichen würde.

  Mittels des „mdate“–Attributs ist für die meisten Publikationen das Datum der letzten Änderung bzw. Aktualisierung zugreifbar. Bei einer Aktualisierung könnte man wie gewohnt die Datei `dblp.xml` durchlaufen, dabei allerdings nur solche Publikationen betrachten, bei denen dieses Datum nach

dem Datum der letzten Indexerstellung oder -Aktualisierung liegt und für diese den Eintrag im Index aktualisieren. Die Erkennung bzw. Zuordnung könnte hier z.B. über den an beiden Stellen erfassten DBLP–Schlüssel erfolgen.

Als etwas schwieriger könnte sich die Behandlung gelöschter Publikationen erweisen, da hierzu vom bestehenden Index ausgehend der Datenbestand mit der Datei `dblp.xml` abgeglichen werden muss; Publikationen, die im Index aber nicht mehr in der Datei vorhanden sind, müssten gelöscht werden. Andererseits stellt sich die Frage, ob diese Funktion überhaupt notwendig ist; im Allgemeinen ist kaum davon auszugehen, dass einmal erfasste Datensätze wieder aus dem Bestand gelöscht werden.

- *Spezieller Analyzer:* Evtl. könnte es sinnvoll sein, eine vollständige (maschinelle) Analyse der Dateninhalte der einzelnen Feldtypen zu machen; es ist zumindest nicht auszuschließen, dass dadurch Fälle ans Licht kommen, in denen der bisher benutzte `StandardAnalyzer`, insb. im Bezug auf die Token–Erzeugung und Stopwörter–Entfernung, nicht optimale Ergebnisse erbringt.

  Denkbar wäre evtl. dass z.B. gewisse Fachbegriffe oder -ausdrücke die Sonderzeichen enthalten aber eigentlich als ein zusammenhängender Term indexiert werden sollen, aufgesplittet werden und deshalb bei der Suche nicht richtig gefunden werden. In einem solchen Fall könnte man darüber nachdenken, eine eigene Analyzer–Klasse zu implementieren, um diese Probleme abzufangen.

- *Überprüfung der UTF–8–Unterstützung bzw. Zeichenkodierung:* Wie bereits im entsprechenden Abschnitt oben angemerkt, stellt die korrekte Kodierung der Zeichenketten bzw. die Umwandlung von und in die von den APIs und Bibliotheken benutzten Formate ein recht komplexes und mitunter schwieriges Thema dar.

  Auch wenn der vorhandene Code beim Testen der Anwendung einwandfrei funktionierte ist nicht auszuschließen, dass dort noch Probleme, Fehler oder Unzulänglichkeiten vorhanden sind. Eine Sichtung durch einen Fachmann in diesem Bereich und evtl. Berichtigung oder Ergänzung hier wäre eine sinnvolle Maßnahme.

- *Berücksichtigung aller Attribute der Datenfelder:* Evtl. könnte man darüber nachdenken, alle Daten, d.h. auch die in den XML–Attributen „href“ von „publisher“ bzw. „series“, „label“ von „cite“ sowie später (wenn entsprechende Daten vorhanden sind) „logo“ von „layout“ enthaltenen Informationen, in den Index mit aufzunehmen.

Aufgrund der Semantik der entsprechenden Felder ist es allerdings eher unwahrscheinlich — noch unwahrscheinlicher als bei den Feldern „url“ und „cdrom“ — dass Nutzer Interesse an einer entsprechenden Suche hätten, auch wenn man damit dann wirklich den *gesamten* DBLP–Datenbestand abbilden würde.

Kapitel 5

Die Suchanwendung

Zumindest aus der Sicht des Endanwenders sind die Nutzeroberfläche mit Eingabeformular für die Abfrage und der Präsentation der Ergebnisse wohl der wichtigste und eigentliche Teil der Suche. Übersichtlichkeit, Schnelligkeit und Qualität der Treffer sind die eigentlichen Kriterien, nach denen er die Anwendung bewertet.

Die Planung und Vorbereitung, die Datenaufbereitung und die bei jeder Abfrage im Hintergrund ablaufenden Prozesse spielen aus dieser Sicht nur eine sehr geringe Rolle, obwohl sie letztendlich maßgeblich dazu beitragen, diese Wünsche des Nutzers zu erfüllen.

In diesem Kapitel werden kurz die allgemeinen Aspekte und Methoden erklärt, die bei der Auswertung von Suchabfragen verschiedener Typen eine Rolle spielen. Anhand der CLucene–Engine werden diese Abläufe wiederum an einer konkreten Implementierung vorgestellt. Schlussendlich wird das Programm `dblpsearch` beschrieben, dass die Umsetzung dieser Punkte für die neue DBLP–Suche darstellt.

5.1 Ablauf von Suchen im Invertierten Index

5.1.1 Boolean–Suche

Wie bereits in Abschnitt 4.1.1 beschrieben, ist die grundlegende Vorgehensweise bei einer Boolean–Suche in einem Invertierten Index recht einfach: Mittels des Index werden die Dokumentmengen, welche die Suchterme enthalten, bestimmt; dann wird, entsprechend der evtl. angegebenen Operatoren, daraus die letztendliche Ergebnismenge gebildet.

Um bei diesem Verfahren, insb. bei komplexeren Abfragen, eine ausreichende

Effizienz zu gewährleisten sind jedoch einige Punkte zu beachten:[1]

- *Auffinden der Termeinträge im Wörterbuch:* Für alle Suchterme muss der entsprechende Eintrag im Wörterbuch schnell aufzufinden sein; ebenso muss schnell herauszufinden sein, falls ein Term *nicht* im Wörterbuch enthalten ist.

  Da im Normalfall die Terme alphabetisch sortiert sind und nur nach exakten Übereinstimmungen gesucht wird, ist das z.B. mit binärer Suche verhältnismäßig einfach zu erreichen. Hierbei spielt natürlich auch eine Rolle wo bzw. wie genau die Daten gespeichert sind, also z.B. ob alle relevanten Informationen im Speicher gehalten werden oder ob ein Nachladen von Festplatte nötig ist.

- *Kombinieren der Dokumentlisten der Einzeltermen:* Bei mehreren Suchtermen, welche implizit oder explizit mit Boolean–Operatoren verknüpft werden, muss die Ergebnismenge effizient erzeugt werden können.

  Da die Dokumentlisten im Normalfall nach einer eineindeutigen Dokument–Id sortiert sind, gibt es Möglichkeiten, die benötigten Operationen (Vereinigung, Schnittmengenbildung, Differenz) recht einfach und effizient zu realisieren.

- *Query–Optimierung:* Um bei komplexen Abfragen die Gesamtarbeit so gering wie möglich zu halten, ist es sinnvoll, die Bearbeitung der einzelnen Terme bzw. Verknüpfungen in einer bestimmten Reihenfolge vorzunehmen.

  Weil z.B. bei einer UND–Verknüpfung von Suchtermen die daraus resultierende Dokumentmenge als Schnittmenge der beiden Term–Dokumentmengen nicht größer sein kann, als die kleinere der beiden Term–Dokumentmengen, macht es hier Sinn, mit dem Term, der in den wenigsten Dokumenten vorkommt zu beginnen. Wenn man in dieser Weise die Verknüpfungen aufsteigend nach Größe der Term–Dokumentmengen durchführt, ist der Aufwand am geringsten.

5.1.2 Ähnlichkeitssuche

In der normalen Boolean–Suche wird im Allgemeinen gefordert, dass die Terme im Index *exakt* mit den eingegebenen Suchtermen übereinstimmen. Bei der Ähnlichkeitssuche dagegen ist nur eine „ausreichend große“ Übereinstimmung erforderlich. Das Maß der gewünschten Ähnlichkeit ist entweder fest von der Suchmaschine vorgegeben oder kann nach Wunsch vom Nutzer selbst festgelegt werden.

[1] vgl. [MRS07], S. 9ff.

Die Ähnlichkeit zweier Zeichenketten wird dabei mittels sog. String–Metriken definiert. Die wohl bekannteste dieser Metriken ist die Levenshtein–Distanz. Sie berechnet das Maß der Ähnlichkeit zweier Zeichenketten über die minimal erforderliche Anzahl von Einfüge-, Lösch- und Ersetzungsoperationen, die nötig sind um die eine Zeichenkette in die andere zu überführen.[2] Andere Metriken berücksichtigen z.B. noch weitere Operationen (z.B. Vertauschung zweier Zeichen) oder weisen den einzelnen Operationen unterschiedliche Gewichtungen zu.

Bei Suche nach exakter Übereinstimmung ist das Auffinden des entsprechenden Termeintrags im Wörterbuch des Index recht einfach mit z.B. einer binären Suche (bei alphabetischer Sortierung, der Normalfall) zu erreichen. Bei der Ähnlichkeitssuche gestaltet sich die Suche nach ausreichend ähnlichen Termen allerdings schwieriger.

Die triviale Vorgehensweise, für alle Terme im Wörterbuch die Ähnlichkeit mit den Suchtermen zu berechnen und dann diejenigen mit ausreichend hohen Werten als Ergebnis zurückzuliefern, ist, selbst bei relativ kleinem Index, natürlich viel zu aufwändig. Stattdessen wird versucht mit speziellen Heuristiken die Menge der in Betracht zu ziehenden Terme im Wörterbuch bereits im Vorfeld auf ein möglichst kleines Maß zu reduzieren und dann nur noch für diese Terme die Abschätzung vorzunehmen.

Hierbei sind teilweise erweiterte Index–Strukturen sinnvoll bzw. erforderlich, z.B. solche in denen auch verschiedene Permutationen oder auch n–grams der Zeichen der Terme verwaltet werden.[3]

5.1.3 Synonymsuche

Die Synonymsuche funktioniert im Normalfall so, dass in der vom Nutzer eingegebenen Abfrage die einzelnen Suchterme automatisch, anhand eines Thesaurus, jeweils um weitere Synonyme oder verwandte Begriffe ergänzt werden („query expansion“). Die entstandene, erweiterte Abfrage wird dann ganz normal ausgewertet.

Die ergänzten Synonyme können dabei aus unterschiedlichen Quellen stammen: ein manuell erstellter Thesaurus, evtl. sogar speziell für das Fachgebiet des Datenbestandes; automatisch, anhand der Wortverteilung in der Dokumentsammlung, erstellte Listen von (wahrscheinlich) verwandten Worten; oder sogar anhand früher benutzter, ähnlicher Abfragen ergänzte Begriffe.

Des weiteren kann diese Art der Suche auch mit weiteren Maßnahmen kom-

[2] vgl. z.B. http://de.wikipedia.org/wiki/Levenshtein-Distanz

[3] vgl. [MRS07], S. 44ff.

biniert werden; z.B. könnte eine unterschiedliche Gewichtung der Terme[4] angewandt werden, mit geringerer Gewichtung für die automatisch ergänzten Begriffe.[5]

5.1.4 Proximity- und Phrasensuche

Die Phrasensuche, d.h. die Suche nach einer gegebenen Folge von Termen, und die Proximitysuche, d.h. die Suche nach einer Menge von Termen in einer bestimmten Entfernung voneinander, können als verhältnismäßig eng miteinander verwandt angesehen werden.

Bei Benutzung einer Edit–Metrik für Terme[6] z.B. könnte die exakte Suche nach einer Phrase als Suche nach einer vorgegebenen Wortmenge mit Distanz 0, d.h. ohne jegliche Änderungsoperationen, implementiert werden.

In diesem Bereich spielt auch die Entfernung von Stopwörtern bei der Indexerstellung eine wichtige Rolle. Je nach Vorgehensweise bei der Phrasensuche könnten gewisse Phrasen nicht gefunden werden, sofern sie ein — durch die Stopwort–Entfernung nicht mehr im Index enthaltenes — Stopwort einschließen.[7] Auch in diesem Fall könnte man z.B. mit dem obigen Verfahren trotzdem gute Ergebnisse erzielen.

Insb. Für diese Art der Suche sind Informationen über die Position der Terme im jeweiligen Dokument unabdingbar (vgl. Abschnitt 4.1.2).

5.1.5 Suche mit Platzhaltern oder Regulären Ausdrücken

Die Suche mit Platzhalten ist in vieler Hinsicht mit der Ähnlichkeitssuche zu vergleichen; im Prinzip handelt es sich ja auch um eine spezielle Form dieser Suchvariante (vgl. Abschnitt 1.4.6).

Aus diesem Grund gestaltet sich die Implementierung ähnlich, wobei auch die gleichen Probleme bzw. Überlegungen auftreten: Die triviale Methode, einfach alle im Index–Wörterbuch enthaltenen Suchterme auf die Übereinstimmung mit dem gegebenen Suchmuster zu überprüfen, ist wiederum viel zu aufwändig. Die bei der Ähnlichkeitssuche angewandten Erweiterungen für das Index–Wörterbuch erweisen sich aus diesem Grunde auch hier als nützlich und ermöglichen eine relativ effiziente Unterstützung dieser Suchart.[8]

[4] vgl. Abschnitt 1.4.5
[5] vgl. [MRS07], S. 149.
[6] Wie z.B. von Lucene angewendet, vgl. Abschnitt 5.2.1.
[7] vgl. [MRS07], S. 24.
[8] vgl. [MRS07], S. 39ff.

5.2 Suchen mittels Lucene

Die wichtigste Lucene–Klasse für den Bereich der eigentliche Suche ist die `Query`–Klasse mit ihren diversen Unterklassen. Diese Klassen repräsentieren die Suchabfrage mit den dafür benötigten Informationen. Eine passend konfigurierte Instanz wird einem `IndexSearcher`[9] übergeben, der dann damit die Abfrage auf dem gewünschten Suchindex auswertet. Die Ergebnisse der Suchabfrage werden als ein Objekt vom Typ `Hits` zurückgeliefert, welches eine nach der ermittelten Score sortierte Liste von Resultaten bereitstellt.

5.2.1 Repräsentation der Suchabfrage

Die verschiedenen von Lucene angebotenen Suchmöglichkeiten werden durch verschiedene Unterklassen von `Query` implementiert. Diese verwalten die zur Abfrage gehörigen Daten, also insb. die entsprechenden Suchterme und evtl. weitere Konfigurationsdaten.[10]

TermQuery Hierbei handelt es sich um die einfachste Abfragemöglichkeit in Lucene; dieser Query liefert alle Dokumente zurück, die den angegebenen Term beinhalten.

BooleanQuery Diese Query–Form kombiniert mehrere andere Queries mittels `BooleanClause`–Instanzen. Diese enthalten jeweils den entsprechenden Teilquery und eine Angabe, ob der entsprechende Teilquery für die Dokumente zutreffen kann (SHOULD; ergibt sich bei Verknüpfung mit OR/ODER), zutreffen muss (MUST; ergibt sich bei Verknüpfung mit AND/UND) oder *nicht* zutreffen darf (MUST NOT; ergibt sich bei Vorstellen von NOT/-NICHT). Bei den Teilqueries kann es sich um beliebige der hier aufgeführten `Query`–Unterklassen handeln.

PhraseQuery Repräsentiert die Suche nach einer Kette von Termen. Diese Query–Form erlaubt eine Art Ähnlichkeitssuche insofern, als das mittels des sog. „Slop"–Faktors angegeben werden kann, in wie weit sich eine im Dokument befindliche Term–Kette von der gegebenen unterscheiden darf. Der Faktor entspricht dabei einer Edit–Distanz–Metrik bei der Einfügen und Bewegen von Termen innerhalb der Kette berücksichtigt werden. Ein Faktor von 0 entspricht dabei einer exakten Suche.

[9] Oder einer anderen `Searcher`–Subklasse; hier relevant ist aber nur `IndexSearcher`.

[10] vgl. [ASFLjd], insb. auch Beschreibungsseite des Packages `org.apache.lucene.search`.

RangeQuery Dieser Query entspricht der Suche nach Dokumenten, bei denen ein Term vorkommt, der innerhalb eines gegebenen Intervalls liegt. Der Vergleich von Termen erfolgt hierbei über die Methode `Term.compareTo` die einen lexikographischen Zeichen–für–Zeichen–Vergleich der beiden Zeichenketten durchführt.

PrefixQuery, WildcardQuery Diese Query–Formen ermöglichen die Suche mittels Platzhaltern. Da der Lucene–Index keine spezielle Unterstützung für diese Suchform anbietet ist die Performance von Platzhaltersuchen, insb. solchen mit Platzhaltern am Anfang oder nahe dem Anfang des Suchterms, verhältnismäßig schlecht, da eine große Anzahl von Termen aus dem Index überprüft werden muss. Da eine Suche mit einem Platzhalter, der sich nur am Ende des Suchterms befindet, aber effizienter realisiert werden kann, existieren aus diesem Grund zwei Implementierungen.

FuzzyQuery Mittels der `FuzzyQuery`–Klasse wird die Ähnlichkeitssuche ermöglicht. Die Ähnlichkeit zweier Terme wird dabei mittels der Levenshtein–Distanz bestimmt.

Neben den oben aufgeführten `Query`–Unterklassen gibt es noch ein paar weiter Varianten, die aber nur zum internen Gebrauch von Lucene bzw. als Basisklassen oben aufgeführter Unterklassen dienen.

5.2.2 Generierung der `Query`–Instanzen

Es gibt zwei Möglichkeiten, die eine Suchabfrage repräsentierende `Query`–Instanz bzw. -Instanzen zu erzeugen:[11]

(1) *„Manuell“:* Hierbei erzeugt die Suchapplikation selbst im Programmcode ein oder mehrere Query–Objekte und kombiniert diese in passender Weise miteinander. Diese Variante bietet sich z.B. an, wenn die Art bzw. Struktur der Abfrage sehr genau feststeht.

Die einfache Autorensuche im DBLP entspricht[12] z.B. einfach einer AND–Kombination von mehreren `TermQueries` in einem `BooleanQuery`. Ähnlich liegt der Fall bei der Verwendung eines komplexeren Formulars auf der Nutzeroberfläche wie z.B. bei der bisherigen Erweiterten Suche im DBLP; auch hier entspricht jedes Feld einem einfachen Sub–Query, die dann programmatisch zu einem Query kombiniert werden können, ohne eine vorherige spezielle Interpretation der Suchterme zu erfordern.

[11] vgl. hierzu auch [ASFLqs], Abschnitt „Overview“.

[12] Wenn man von der dortigen Sonderbehandlung von Umlauten und Groß-/Kleinschreibung absieht.

(2) *Mittels* `QueryParser`*:* Hier wird die Suchabfrage als Zeichenkette mit festgelegter Syntax[13] angegeben. Mittels der Klasse `QueryParser` können dann aus dieser Spezifikation die entsprechenden `Query`–Objekte generiert werden. Diese Variante eignet sich bei Benutzung einer einfachen Nutzeroberfläche mit nur einem Eingabefeld zur Eingabe der Suchabfrage als Zeichenkette, wie sie von der neuen DBLP–Suche verwendet wird.

Zu beachten ist hierbei allerdings, dass der `QueryParser` nicht alle Features der `Query`–Klassen unterstützt: So erlaubt die `WildcardQuery`–Klasse an sich die Angabe eines Platzhalters auch am Anfang eines Suchmusters; Dies hat jedoch eine extrem schlechte Performance zur Folge, da diese Form der Abfrage bedeutet, dass *alle* Terme des Index auf Übereinstimmung überprüft werden müssen. Aus diesem Grund wird diese Möglichkeit daher vom `QueryParser` nicht bereitgestellt.

5.2.3 Auswertung der Suchabfrage

Nachdem das die Suchabfrage repräsentierende `Query`–Objekt erzeugt wurde, wird es mittels der `search`–Methode einer Instanz der Klasse `IndexSearcher` für den gewünschten Suchindex übergeben. Der `IndexSearcher` stellt das eigentliche „Frontend“ der Such–API dar und ist — neben den `Query`–Klassen — im Normalfall die einzige Klasse, welche der Anwendungsprogrammierer zum Durchführen einer Abfrage kennen muss.

Intern erzeugt der `IndexSearcher` für den übergebenen `Query` eine entsprechende Instanz vom Typ `Weight`; dabei handelt es sich um eine spezielle Repräsentation des Queries. Dies dient dem Zweck, Änderungen an den `Query`–Instanzen zu vermeiden, insb. um diese auch mehrmals verwenden zu können; sämtliche Searcher–spezifische Zustandsinformationen werden stattdessen in diesem `Weight`–Objekt gespeichert.

Über diese `Weight`–Instanz wird weiterhin ein Objekt vom Typ `Scorer` kreiert. Diese Klasse erlaubt, über die zum `Query` passenden Dokumente zu iterieren und für jedes eine entsprechende Bewertung („Score“) zu berechnen (s.u.).

Die `score`–Methode des `Scorer` führt nun genau diese Aktionen für alle entsprechenden Dokumente aus. Für jedes gefundene, passende Dokument wird mittels des `collect`–Callbacks der Klasse `HitCollector` diesem die Möglichkeit gegeben, das Dokument mit seiner Bewertung in die von ihm verwaltete Ergebnisliste aufzunehmen.

Im Normalfall wird dieser `HitCollector` wiederum in einem Objekt der Klasse `Hits` gekapselt, das eine nach der ermittelten Score sortierte Liste von Resul-

[13] siehe [ASFLqs].

taten verwaltet und einen einfacheren Zugriff auf diese ermöglicht.

5.2.4 Bewertung der Suchergebnisse

Die Bewertung der Trefferqualität für ein Dokument in Lucene („Score“) folgt größtenteils der Vorgehensweise beim Vektorraum–Modell (→ 4.2.1). Die Ähnlichkeit zwischen dem Vektor der Abfrage und dem Vektor des aktuellen Dokuments wird berechnet; je größer die Ähnlichkeit ist, desto höher wird das Dokument bewertet.

Die Berechnung der Ähnlichkeit erfolgt dabei unter Verwendung der Klasse `Similarity`.[14] In der Standard–Implementierung wird hier das Cosinus–Maß benutzt, d.h. es wird der Cosinus des Winkels zwischen Dokument- und Abfrage–Vektor ermittelt. Im Einzelnen berechnet sich für einen gegebenen Query q die Score für ein Dokument d wie folgt:

$$\text{score}(q,d) \quad = \quad \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} \text{score}(t,d) \tag{5.1}$$

mit

$$\text{score}(t,d) = \text{tf}(t,d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d) \tag{5.2}$$

$\text{tf}(t,d) = \sqrt{\text{Häufigkeit}}$ bezeichnet hierbei die Termhäufigkeit („term frequency“), d.h. die Anzahl der Vorkommen des Terms in d. Je öfter also ein Term in einem Dokument vorkommt, desto höher wird die Score.

$\text{idf}(t) = 1 + \log(\#\text{Dokumente}/(\#\text{Vorkommen} + 1))$ steht für die Umgekehrte Dokumenthäufigkeit („inverted document frequency“), d.h. der Kehrwert der Anzahl der Dokumente, in denen der Term zu finden ist. Hier wird davon ausgegangen, dass häufig vorkommende Terme eher wenig zur Unterscheidung von relevanten und irrelevanten Dokumenten beitragen.

t.getBoost() steht für einen evtl. bei der Erzeugung des Queries angegebenen, expliziten Boost–Faktor für den Term t. $\text{norm}(t,d)$ fasst eine Reihe von bei der Indexerstellung festgelegten Werten, insb. explizit festgelegte Boost–Faktoren für das Dokument oder spezielle Felder.

$\text{coord}(q,d)$ ist ein Faktor, der davon beeinflusst wird, wieviele der Suchterme insgesamt in d vorkommen. Unter der Annahme, das Dokumente, in denen viele der Suchterme vorkommen interessanter sind, als solche die nur wenige der Terme enthalten, wird hierdurch die Score zusätzlich zu den Auswirkungen von $\text{tf}(t,d)$ angepasst.

[14] vgl. hierzu und zur Berechnung allgemein die entsprechende Seite in [ASFLjd], sowie auch [ASFLsc].

queryNorm(q) ist nötig, um die für Teilqueries berechnete Bewertung in Bezug auf die Gesamtbewertung der Abfrage vergleichbar zu machen. Der genaue Wert dieses Faktors wird aus diesem Grund auch u.A. von dem jeweiligen `Weight`–Objekt mit beeinflusst.

Da die Berechnung der Ähnlichkeit verhältnismäßig aufwändig ist, wird nicht für jedes von einem (Teil–)Query zurückgelieferte Dokument die Score berechnet, sondern es werden nur solche Dokumente berücksichtigt, die auch in der endgültigen Ergebnismenge auftauchen.

5.3 Das Programm `dblpsearch`

Das Programm `dblpsearch` stellt die eigentliche Nutzerschnittstelle der neuen DBLP–Suche dar. Es handelt sich um eine Webanwendung, die den Nutzern die Eingabe von Abfragen ermöglicht, diese mittels der unterliegenden CLucene–Suchengine auf dem aus dem DBLP–Datenbestand generierten Suchindex auswertet und die gefundenen Ergebnisse wiederum im Web präsentiert.

5.3.1 Bedienung

Die Anwendung ist mit jedem handelsüblichen Webbrowser zu bedienen. Sie verzichtet auf die Benutzung von Java, JavaScript, Active–X und sonstigen Zusatztechnologien und beschränkt sich auf die Benutzung von reinen HTML–Seiten und Webformularen.

Auch in Bezug auf die Nutzeroberfläche ist die Anwendung recht einfach gehalten: Es gibt lediglich zwei Seiten, die beide ebenfalls recht schlicht gehalten sind. Die bestehenden HTML–Seiten des DBLP werden durch entsprechende Verlinkungen aus der Ergebnisliste ebenfalls mitbenutzt bzw. eingebunden.

Die Such- und Ergebnis–Seite

Das Formular zur Eingabe der Suchabfragen ist sehr einfach gehalten: Es besteht lediglich aus einem Textfeld, in dem der Nutzer seine Abfrage eintragen kann und einen Knopf zum Abschicken.

Nach dem Abschicken und der Auswertung der Abfrage werden die Ergebnisse, bzw. ggf. die Meldung, dass keine passenden Datensätze gefunden wurden, ebenfalls auf dieser Seite angezeigt. Außerdem wird der alte Query erneut in das Suchfeld eingetragen. Auf diese Weise ist es schnell und einfach möglich, bei Bedarf erneut eine leicht abgewandelte Abfrage zu stellen.

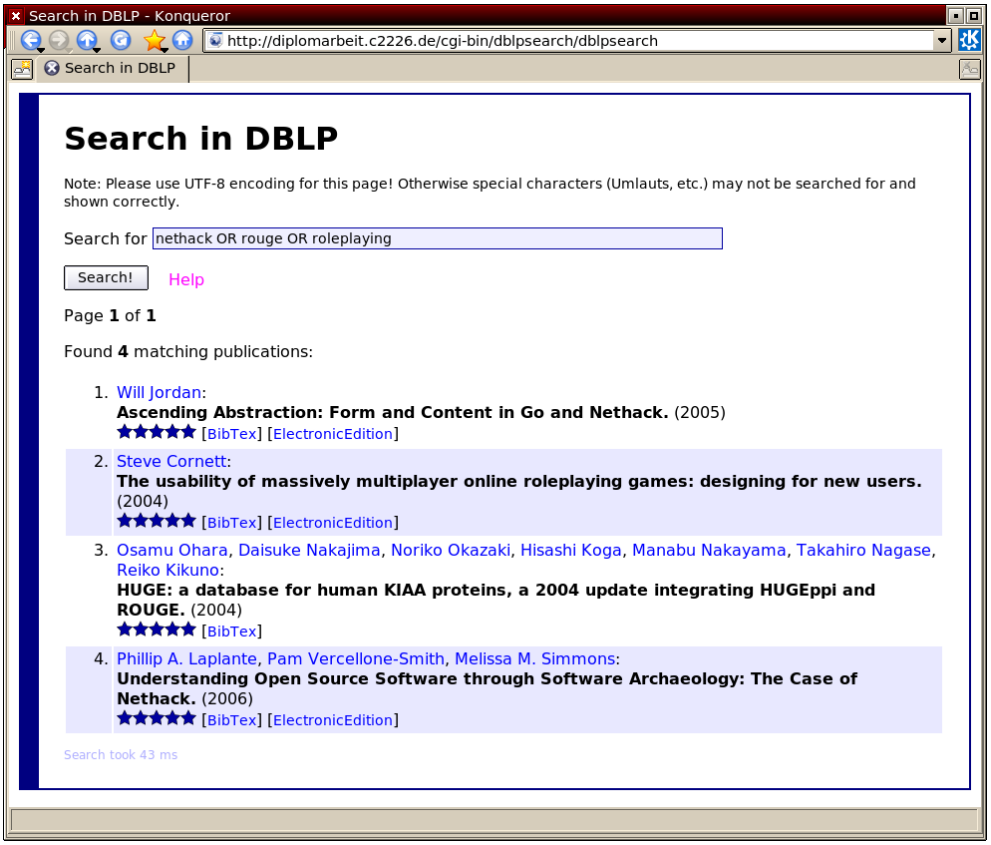

Abbildung 5.1: Suchmaske und Ergebnisliste der neuen DBLP–Suche

Die Treffer in der Ergebnisliste sind nach der von CLucene zugewiesenen Bewertung („Score“, s.o.) absteigend sortiert (d.h. die am besten passenden Treffer werden am Anfang der Liste angezeigt) und entsprechend durchnummeriert.

Die einzelnen Angaben zu jedem Treffer sind, der Übersichtlichkeit halber, auf relativ wenige Punkte beschränkt. Ausgegeben werden:

- Die Autoren der Publikation. Die Namen sind jeweils mit der entsprechenden Autorenseite auf der DBLP–Website verlinkt.
- Der Titel der Publikation und das Erscheinungsjahr werden darunter aufgeführt.
- Zur Klassifizierung wie gut ein Treffer zur Suchabfrage passt, werden 1 (passt kaum) bis 5 (passt sehr gut) Sterne angezeigt. Der von CLucene gelieferte Score–Wert wird nicht direkt angezeigt, da der genaue Wert im Prinzip keine weitere große Aussagekraft hat und diese graphische Art der Anzeige

schneller und einfacher zu erfassen sein dürfte.[15]

- Ein Verweis auf die BibTEX–Seite im DBLP ermöglicht, direkt weitere Informationen zur Publikation zu erhalten bzw. die entsprechenden Daten direkt in eine eigene Bibliographie aufzunehmen.

- Wenn für die Publikation ein Link zu einer „Electronic Edition“ vorhanden ist, wird dieser ebenfalls angezeigt.

Da die genaue Spezifikation der Suchabfrage über die Lucene–Query–Sprache im Text erfolgt, sind hierfür keine weiteren Steuerungselemente nötig.

Die Anzeige der Suchergebnisse erfolgt seitenweise, mit einem von der Anwendung vorgegebenen, vom Nutzer nicht konfigurierbaren Limit von (max.) 10 Treffern pro Seite.

Die Hilfe- und Statistik–Seite

Diese von der Suchseite verlinkte Ansicht bietet dem Nutzer Hilfestellung zur Benutzung der Suche sowie einige Statistiken zum Suchindex bzw. zum Datenbestand selber. Angezeigt werden:

- Informationen zur verwendeten Basis–Suchmaschine (CLucene) und deren Version; dies ist insb. auch insofern interessant, als dass gewisse Features bzw. Suchmöglichkeiten erst in neueren Versionen der Engine zur Verfügung stehen. In der bei dieser Arbeit verwendeten Version 0.9.18 ist es z.B. noch nicht möglich, die gewünschte max. Edit–Distanz bei der Ähnlichkeitssuche manuell zu spezifizieren.

- Das Datum der letzten Index–Aktualisierung bzw. -Erstellung um die Aktualität des durchsuchten Datenbestandes zu kommunizieren.

- Die Anzahl der indexierten Dokumente und im Wörterbuch enthaltenen Terme; dies ist für die Benutzung eher unwichtig und wird eigentlich nur der Vollständigkeit halber aufgeführt :-)

- Wichtiger dagegen ist die Liste der verfügbaren Datenfelder. Anhand dieser Auflistung kann der Nutzer erkennen, welche Daten(felder) verfügbar sind, d.h. nach welchen Daten aus dem DBLP–Datenbestand gesucht werden kann und wie diese anzusprechen bzw. in der Abfrage anzugeben sind.

- Die Syntax der Abfragesprache kann auf der entsprechenden offiziellen Lucene–Webseite eingesehen werden, zu der hier ein Link gesetzt ist.

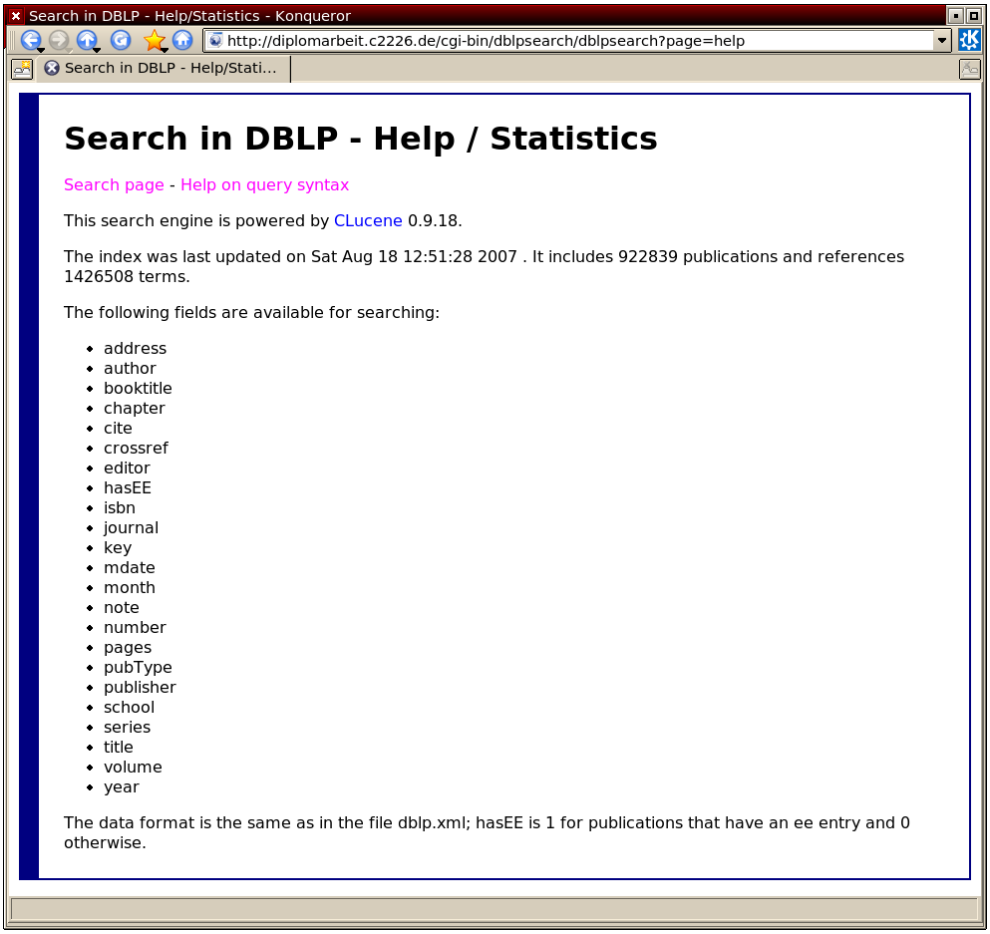

Abbildung 5.2: Hilfe- und Statistikseite der neuen DBLP–Suche

Die auf dieser Seite befindlichen Statistiken zum Index (Datum der Aktualisierung, Anzahl Dokumente und Terme, Feldliste) sind nicht fest kodiert sondern werden aus dem aktuellen Suchindex ausgelesen und dynamisch auf der Seite ausgegeben.

5.3.2 Programmarchitektur und -ablauf

Allgemeine Request–Behandlung

Da es sich bei `dblpsearch` noch um eine verhältnismäßig kleine Applikation handelt, wurde auf den Einsatz eines externen Web–Applikation–Frameworks verzichtet und stattdessen eine Handvoll eigener Klassen entwickelt, die auch ohne

[15] vgl. auch [ASFLfaq], Abschnitt „Can I filter by score?“

großen Overhead eine einigermaßen saubere und klare Struktur für die Anwendung ermöglichen.

Eine vom Nutzer übermittelte Anfrage, sei es nach einer Seite oder die Übertragung einer Suchabfrage, löst über das Modul `dblpsearch`, dass nur einen sehr dünnen Wrapper darstellt, einen Aufruf der Methode `dispatch` an einem (eigens neu erstellten) Objekt der Klasse `Dispatcher` aus.

Diese Methode wertet den Parameter „page“ aus, der angibt, welche der unterstützten Seiten („index“ — die Such- und Ergebnis–Seite oder „help“ — die Hilfe- und Statistik–Seite) angezeigt werden soll. Der Standard, falls der Parameter nicht angegeben wurde, ist „index“.

Die eigentliche Behandlung der Abfrage bzw. die Generierung und Anzeige der Seite wird dann von einer entsprechenden `PageHandler`–Unterklasse durchgeführt. Sollte ein Fehler auftreten wird dieser vom `ErrorPageHandler` an den Nutzer weitergegeben.

Die Handler–Klassen

Jede Seite der Applikation wird von einer speziellen `PageHandler`–Unterklasse verwaltet. `PageHandler` stellt eine Methode `show` bereit, die wiederum die drei Methoden `showHeader`, `showContents` und `showFooter` aufruft. Die Methode `showContents` ist dabei „pure virtual“, d.h. muss in jedem Fall von einer konkreten Unterklasse implementiert werden, da sie für die Anzeige des speziellen Inhalts der jeweiligen Seite verantwortlich ist. Die Standard–Implementierung der beiden anderen Methoden gibt die grundlegenden Kopf- bzw. Fußinformationen für HTML–Seiten (`html` und `body`–Tags) aus; alternativ können auch zwei Dateien angegeben werden, aus denen entsprechende HTML–Fragmente geladen werden.

Abgeleitet von `PageHandler` ist die Klasse `FormHandler` welche zusätzlich noch eine Methode `handle` bereitstellt. `FormHandler` dienen, wie am Namen bereits zu sehen ist, zur Verwaltung von Seiten mit Formularen, d.h. Möglichkeiten für Nutzereingaben. Die Bearbeitung dieser Nutzereingaben sollte innerhalb der `handle`–Methode erfolgen.

Die Suche und Ergebnisanzeige – `TextQueryFormHandler`

Die wohl wichtigste Klasse für die Nutzeroberfläche ist `TextQueryFormHandler`. Sie ist für die Anzeige des Suchformulars, die Auswertung der Suchabfragen und die Anzeige der Ergebnisliste zuständig.

In der `handle`–Methode wird überprüft, ob der Knopf zum Abschicken des

Query–Strings gedrückt wurde. Wenn ja, wird die Zeichenkette aus den von `cgicc` übermittelten Request–Informationen ausgelesen und an das CLucene–Backend weitergeleitet. Die Ergebnisse werden entgegengenommen und in einer Klassenvariable für die weitere Ausgabe zwischengespeichert.

Die `showContents`–Methode ruft wiederum zwei weitere Methoden namens `showForm` und, falls Resultate einer Suche vorliegen, `showResults` auf.

`showForm` sorgt für die Ausgabe des HTML–Codes zur Anzeige des Suchformulars, evtl. mit Eintragen einer evtl. bereits von früher vorhandenen Query–Zeichenkette.

`showResults` gibt, mit Hilfe des zwischengespeicherten `Hits`–Objekts, die Ergebnisliste aus. Die Anzeige jedes einzelnen Treffers wird dabei von der Klasse `RecordResultRepresentation` übernommen, die das genaue Aussehen steuert. Durch Austausch dieser Klasse gegen eine andere Implementierung kann so die Anzeige der Treffer in der Liste leicht veränderten Anforderungen angepasst werden.

Zur Generierung der Links zu den Autoren- und BibTEX–Seiten der DBLP–Website kommen Unterklassen von `URLConverter` zum Einsatz. Diese erzeugen aus den als Methodenargumenten übergebenen Werten, hier dem Namen des Autors für `Authorname2URLConverter` bzw. dem DBLP–Schlüssel der angezeigten Publikation für `PubKey2URLConverter`, eine gültige URL im passenden Format.

5.3.3 Performance

Um die Geschwindigkeit der Anwendung, insb. unter Last, abschätzen zu können, wurden eine Reihe von Tests durchgeführt. Unter Verwendung eines `bash`[16]–Shellskripts wurden mit Hilfe des Programms `wget`[17] (mehr oder weniger) parallel jeweils 50 Suchabfragen an die Applikation gesendet. Aus den erhaltenen HTML–Seiten wurde die dort angezeigt Bearbeitungsdauer für die Suchanfrage extrahiert und protokolliert. Um die Performance–Messung dabei nicht durch die von den `wget`–Instanzen verbrauchte Leistung zu beeinflussen, wurden die Abfragen über ein lokales Netz von einem zweiten Rechner aus verschickt.

Die erhaltenen Ergebnisse für vier „einfache“ Abfragen sind in Abbildung 5.3 dargestellt. Bei den vier Queries handelt es sich um eine exakte Suche nach einem Term (lieferte 14096 Treffer), um eine Prefix–Suche (d.h. Wildcard–Suche mit nur einem Platzhalter am Ende des Suchterms; 21717 Treffer), um eine einfache Boolean–Suche mit zwei einfachen Suchtermen (13 Treffer) sowie um eine

[16] Eine bekannte Kommandozeilenumgebung; siehe http://www.gnu.org/software/bash/

[17] Ein Kommandozeilen–Programm zum Herunterladen von Webseiten/Dateien u.A. per HTTP–Protokoll; siehe http://www.gnu.org/software/wget/.

Boolean–Suche bei der der zweite Term Platzhalter enthält (302 Treffer).

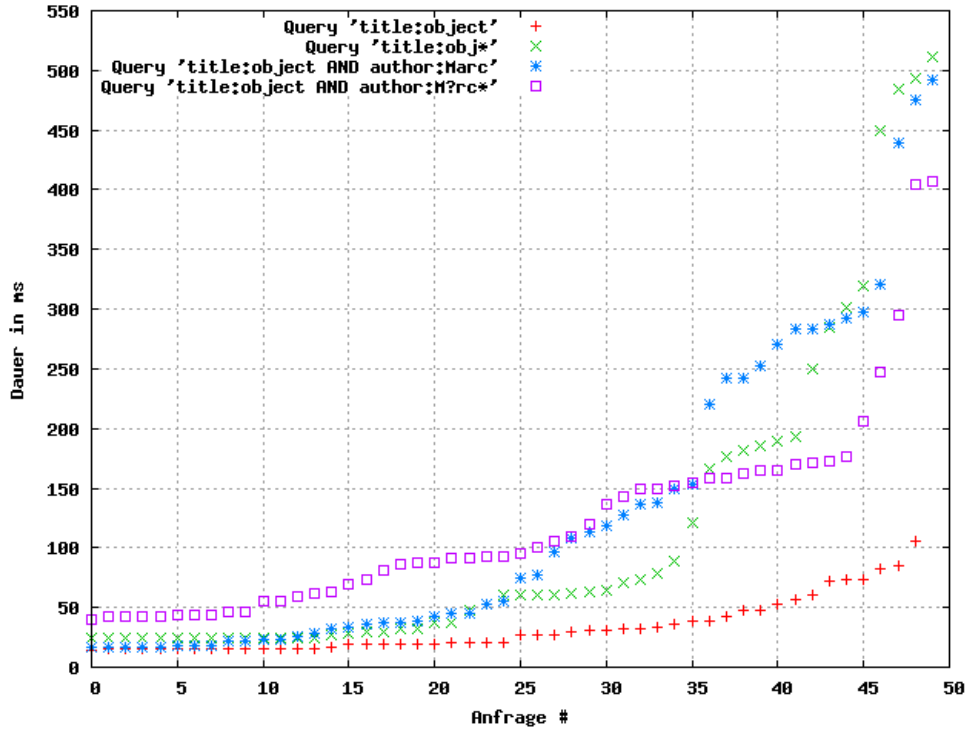

Abbildung 5.3: Gesamtdauer verschiedener einfacher Queries, parallele Anfragen

Wie man gut erkennt, verkraftet das System offensichtlich eine gewisse Menge von gleichzeitigen Anfragen recht gut; für die ersten zehn Abfragen ist die Antwortzeit relativ konstant niedrig. Bis zur 25. Abfrage verdoppelt sie sich ungefähr, was aber in Anbetracht der immer noch im Zehntelsekunden–Bereich liegenden Werte in der Praxis kaum auffallen dürfte. Erst danach steigen die Werte deutlicher und im Bereich um die 40.–45. Abfrage ist die Verlängerung der Suchdauer dann sehr auffällig — ein Faktor von 20 gegenüber der Zeit der ersten Anfragen ist hier teilweise sichtbar.

Nichtsdestotrotz liegen aber selbst die höchsten Werte hier immer noch lediglich im Bereich einer halben Sekunde; verglichen mit der bisherigen Suche ist das immer noch ein sehr guter Schnitt.[18]

Zum Vergleich wurden diesselben Tests auch sequentiell durchgeführt. Statt die `wget`–Instanzen parallel nebeneinander laufen zu lassen, wurden diese hierbei

[18] Wobei hier selbstverständlich berücksichtigt werden muss, dass es sich beim bisherigen DBLP–Server evtl. um eine wesentlich schwächere Maschine handelt; insofern kann man die Werte nur sehr bedingt miteinander vergleichen.

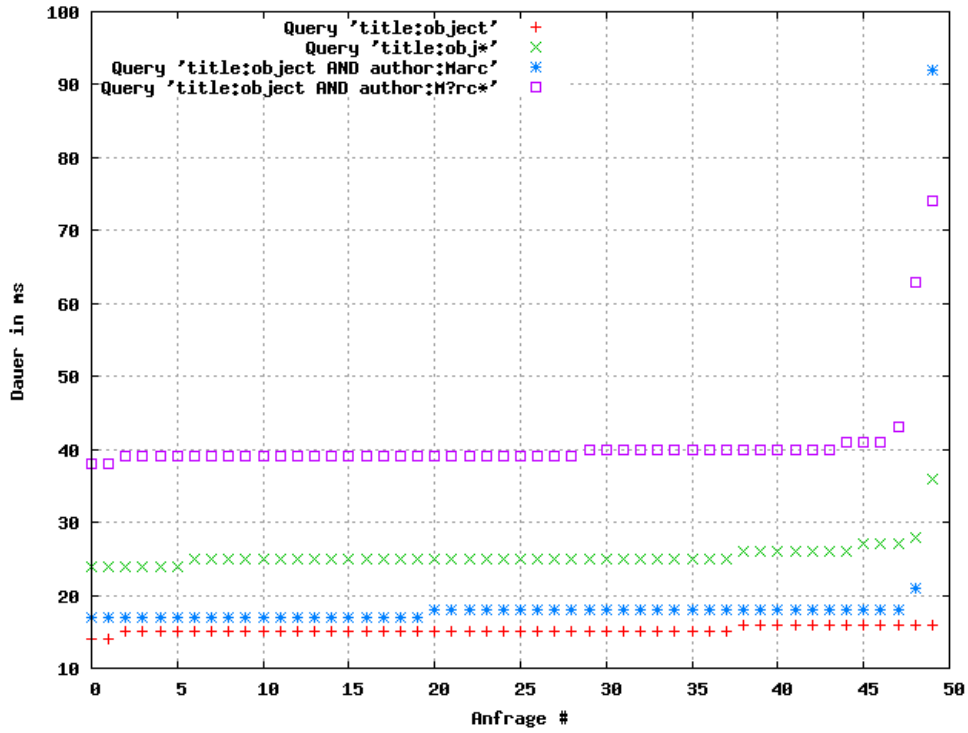

Abbildung 5.4: Gesamtdauer verschiedener einfacher Queries, sequentielle Anfragen

nacheinander ausgeführt, so dass jetzt immer nur eine Abfrage zur Zeit behandelt werden musste. In Abbildung 5.4 erkennt man jetzt deutlich die annähernd konstante Bearbeitungsdauer. Der immer noch vorhandene, leichte Anstieg der Zeiten ist vermutlich auf die notwendigen „Aufräumarbeiten“ zurückzuführen, welche für die bei der Bearbeitung der vorherigen Anfrage allozierten Ressourcen durchgeführt werden. Der auffällige hohe Anstieg bei den letzten Abfragen ist evtl. auf ein internes Limit des Webservers zu begründen, das z.B. nach einer gewissen Anzahl von Abfragen größere Verwaltungsarbeiten nötig macht.

Um die Leistung bei komplexeren Aufgaben beurteilen zu können, wurden weiterhin zwei aufwändigere Abfragen abgesetzt. Die Aufrufe erfolgten ebenfalls parallel. Die Ergebnisse sind in Abbildung 5.5 dargestellt. Beim ersten Query handelt es sich um eine Ähnlichkeitssuche mit einem Term (lieferte 20301 Treffer), beim zweiten noch ergänzt um einen Term mit Platzhaltern (426 Treffer).

Besonders deutlich ist hier natürlich die schon bei den ersten Anfragen wesentlich höhere Antwortzeit; für den zweiten Query liegt sie hier bereits bei der zweiten Abfrage bei fast zwei Sekunden. Aber auch hier ist, sehr ähnlich wie in den vorherigen Fällen, ein eher leichter Anstieg bis ungefähr zur 35.–40. Abfrage zu erkennen; dort verlängern sich die Suchdauern dann wiederum erheblich bis

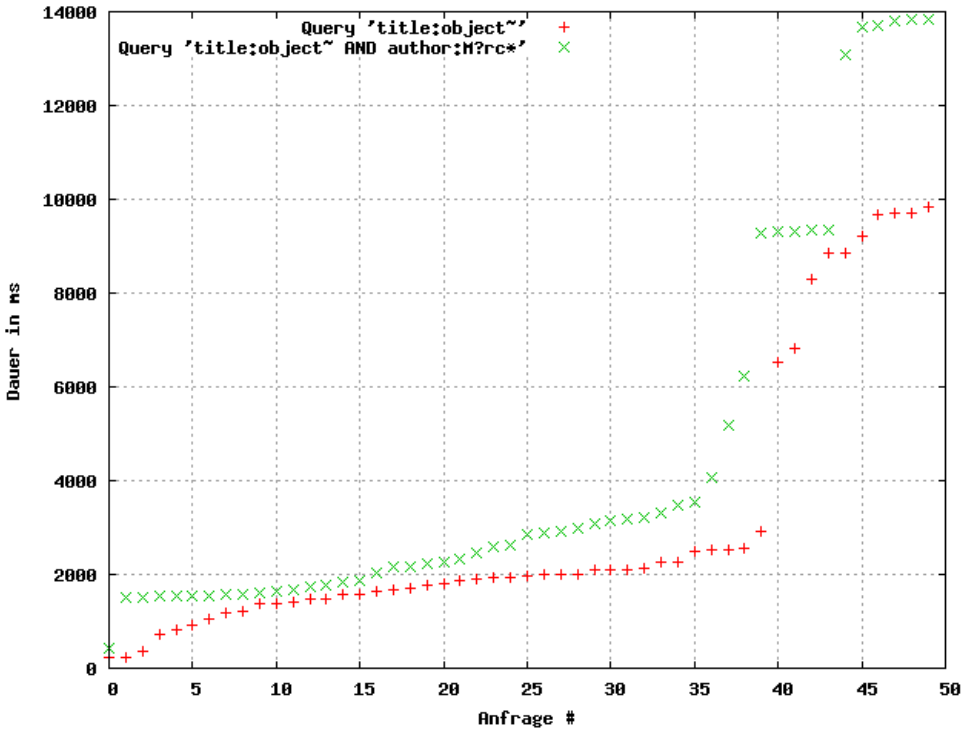

Abbildung 5.5: Gesamtdauer komplexerer Queries, parallele Anfragen

auf Spitzenwerte von fast 10 bzw. fast 14 Sekunden für die letzten Abfragen.

Die Umstellung auf nacheinander erfolgende Aufrufe (siehe Abbildung 5.6) gleicht auch hier der entsprechenden Situation der einfachen Queries. Die Bearbeitungsdauer ist über alle Anfragen weitgehend konstant, mit nur einem sehr geringen Anstieg. Auffällig, wenn auch mit nur einem „Ausreißer“ wesentlich weniger ausgeprägt, ist auch hier wieder der starke Anstieg bei der letzten Anfrage.

Selbstverständlich sind diese Tests in keiner Weise ausreichend, um wirklich genaue Aussagen über die Performance der Anwendung zuzulassen; hierzu wären mehr und tiefergehende Versuche nötig. U.a. spielen einfach zu viele Faktoren eine Rolle, angefangen von Dingen wie der Startgeschwindigkeit der `wget`–Instanzen (die über die Parallelität der Anfragen entscheiden) über die aktuelle Systemauslastung bis zur den genauen Anfangs- und Endzeiten der Bearbeitung der einzelnen Anfragen durch den Webserver.

Auch ist zu beachten, dass es sich bei den betrachteten Zeiten nur um die Bearbeitungsdauer für die eigentliche Suchanfrage innerhalb der CLucene–Suchmaschine handelt. Weitere Faktoren, wie die zum Starten des Anwendungsprozesses benötigte Zeit, die Dauer der Datenübertragung oder der Aufbau der Seite sind

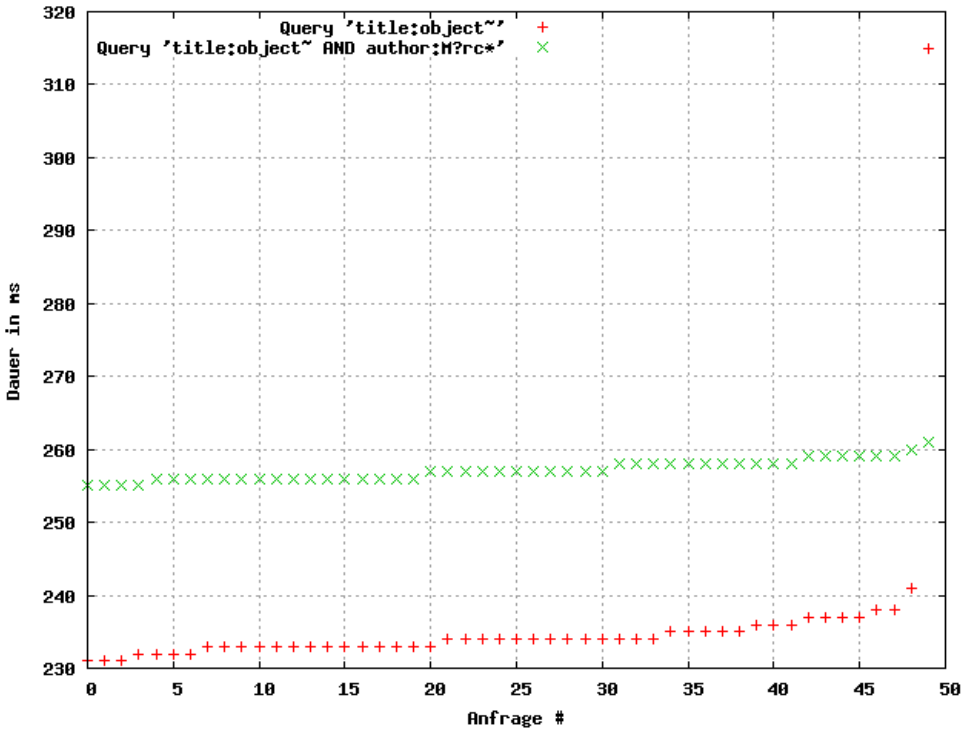

Abbildung 5.6: Gesamtdauer komplexerer Queries, sequentielle Anfragen

hier ebenfalls nicht berücksichtigt, obwohl sie für den Nutzer natürlich gleichfalls eine wichtige Rolle spielen.

Insgesamt lässt sich aber doch sagen, dass die Leistungsfähigkeit der Anwendung, selbst bei stärkerer Auslastung, durchaus akzeptabel ist. Zwar liegen keine Statistiken über die genaue Nutzeranzahl der DBLP–Website vor, aber vermutlich sind auch nur zehn mehr oder weniger gleichzeitige Abfragen eher unwahrscheinlich, so dass die vorhandenen Leistungswerte der Suche im Allgemeinen vollkommen ausreichend sein dürften.

5.3.4 Todos und Verbesserungsmöglichkeiten

Die `dblpsearch`–Suchanwendung stellt, insb. im Vergleich mit den bis dahin verfügbaren Optionen des DBLP, bereits eine recht mächtige aber trotzdem einfach zu bedienende Suchmöglichkeit dar. Nichtsdestotrotz gibt es sicherlich noch Raum für Verbesserungen oder Erweiterungen; beispielsweise:

- Programmiertechnisch ist es zwar einfach möglich, verschiedene Ausgabeformate für die Trefferliste zu erzeugen; weitere Alternativen neben dem

Standardformat wurden bisher allerdings nicht implementiert und es existiert auch keine Möglichkeit für den Nutzer das gewünschte Format auszuwählen.

Als Optionen denkbar wären evtl. eine kompaktere, übersichtlichere Ausgabe der Treffer, z.B. nur mit Autorennamen und Publikationstitel. Umgekehrt könnte andererseits eine noch erweiterte Anzeige von Informationen, mit z.B. Quellenangaben wie Konferenz- bzw. Zeitschriftenname, die direkte Einschätzung der Relevanz eines Treffers für den Nutzer vereinfachen.[19]

Auch die Ausgabe von Treffern in Formaten für den Import in andere Software, wie z.B. EndNote[20] oder RIS[21], sind denkbar. Hier wäre allerdings zu überlegen, ob solche Dinge die Ergebnisliste nicht bereits zu groß und unübersichtlich werden lassen; sinnvoller könnte es sein, stattdessen die Publikationsseiten auf der DBLP–Website, die bisher nur einen BibTEX–Datensatz enthalten, dahingehend zu ergänzen.

- Als alternative Eingabe- bzw. Spezifikationsmöglichkeit für Suchabfragen könnte man ein komplexeres Formular, wie z.B. bei der bisherigen Erweiterten Suche, anbieten.

  Vorteile würde das insofern bringen, als dass man für jedes verfügbare Datenfeld bereits ein Eingabefeld für den oder die Suchterme vorsehen könnte; so müsste der Nutzer die Liste der Felder nicht nachschauen bzw. im Kopf haben. Außerdem könnte man dann Steuerungsmöglichkeiten für die Suche, wie z.B. Aktivieren von Ähnlichkeits- statt exakter Suche, ebenfalls einfach über Kontrollelemente, wie z.B. Checkboxen, realisieren (vgl. Abschnitt 3.3.2).

  Andererseits würde das Formular aufgrund der großen Anzahl von Datenfeldern im DBLP ziemlich groß und evtl. unübersichtlich werden, insb. dadurch, dass die verschiedenen Suchmöglichkeiten für jedes Feld zur Verfügung stehen und meist auch Sinn machen; da die entsprechenden Kontrollelemente also für jedes Feld aufgeführt werden müssten, würde das die Übersichtlichkeit noch weiter einschränken.

- *Erweiterte Navigation in der Ergebnisliste:* Zur Zeit ist in der Ergebnisliste nur ein einfaches Blättern zur vorhergehenden bzw. nächstfolgenden Seite

[19] Allerdings wäre dafür auch die Aufnahme der zusätzlichen Feldinhalte in den Index nötig, vgl. Abschnitt 4.3.2

[20] Ein bekanntes Literaturverwaltungsprogramm, siehe z.B. http://en.wikipedia.org/wiki/EndNote oder die Homepage http://www.endnote.com

[21] Ein Format für die Zitatverwaltung, siehe z.B. http://en.wikipedia.org/wiki/RIS_(file_format)

möglich. Interessant wäre evtl., insb. bei einer großen Anzahl von Seiten, die Option, direkt zu einer ausgewählten Seite zu springen.

Andererseits ist aber im Normalfall wohl davon auszugehen, dass aufgrund der Sortierung nach Relevanz ein Nutzer die Ergebnisse der Reihe nach sichten wird, was den aktuellen Navigationsmöglichkeiten entspricht. Aus diesem Grund ist die Priorität einer solchen Option eher gering und wurde aufgrund von erhöhtem technischen Aufwand nicht implementiert.

Kapitel 6

Fazit

6.1 Zusammenfassung

Das Ziel dieser Arbeit bestand darin, eine zusätzliche Suchmöglichkeit für das DBLP zu entwerfen und implementieren. Dieses Ziel wurde weitestgehend erreicht.

Die Untersuchung der bisher verfügbaren Suchmöglichkeiten des DBLP zeigte, dass diese eine Reihe von unterschiedlichen Schwächen und Unzulänglichkeiten aufweisen. Darüber hinaus gibt es weiterhin verschiedene zusätzliche Suchtypen, die für eine Benutzung, gerade auch im DBLP, als sehr sinnvoll erscheinen. Insb. eine Ähnlichkeitssuche für Autorennamen ist aufgrund der immer noch vorhandenen, unterschiedlichen Schreibweisen im DBLP als nützlich zu bewerten.

Bei der Analyse des DBLP–Datenbestandes ergab sich, dass die XML–Datei `dblp.xml`, u.A. aufgrund der klaren Struktur in der die Daten dort abgelegt sind, sowie deren Aktualität und Vollständigkeit, sich als Basis für die Erstellung eines Suchindex am Besten eignet. Die weitere Auswertung der Daten resultierte in einigen Kriterien die eine Suchmaschine erfüllen, bzw. Features die sie unterstützen muss. Hierzu zählen insb. die Möglichkeit, strukturierte, d.h. in verschiedene Felder aufgeteilte Informationen zu verwalten.

Anhand der bisher ermittelten Kriterien wurden mehrere allgemeine Suchmaschinen verglichen, um die als Basis für die neue Suchanwendung geeignetste zu ermitteln; die Wahl fiel dabei schließlich auf CLucene. Diese frei verfügbare Suchmaschine unterstützt eine große Anzahl von Suchtypen (Boolean–Suche, Ähnlichkeitssuche, Wildcard–Suche, Phrasen- bzw. Proximity–Suche,. . .) und die Verwaltung von strukturierten Daten. Daneben bietet sie eine gute Performance und eine einfach zu benutzende Programmierschnittstelle.

Als zu verwendende Programmiersprache wurde C^{++} gewählt. Während der

Arbeit stellte sich allerdings heraus, dass C^{++} im Vergleich mit den anderen zur Auswahl stehenden Sprachen (Java, Perl, PHP) teilweise noch Defizite aufweist. Trotz einiger Verbesserungen der Sprache in der jüngeren Zeit ergaben sich bei der Implementierung der Anwendung, insb. im Bereich der Verarbeitung von Zeichenketten, eine Reihe von Schwierigkeiten. Diese machten z.T. einige zusätzliche Überlegungen und Programmierung nötig. Da diese Probleme jedoch teilweise erst in einem fortgeschrittenen Stadium der Arbeit zu Tage traten, wurde davon abgesehen, nachträglich auf eine andere Sprache zu wechseln.

Als wichtigste Datenstruktur im Bereich der Suche nach textuellen Daten ist der Invertierte Index zu nennen. Diese Datenstruktur verwaltet die in einer Datensammlung vorkommenden Terme („Wörter“) und entsprechende Mengen von Dokumenten, in denen diese Terme vorkommen. Durch diese Struktur wird eine effiziente Suche nach Begriffen ermöglicht. Durch Hinzufügen von Informationen zu Termhäufigkeit und -positionen kann dies noch auf andere Suchtypen (Ranked Queries, Phrasensuche, etc.) erweitert werden. Am Beispiel des Lucene–Index wurde eine konkrete Implementierung eines solchen Invertierten Index vorgestellt.

Bei der Entwicklung des Programms `dblp2index` zur Generierung des Suchindex aus dem DBLP–Datenbestand stellte sich heraus, dass die Erstellung eines solchen Index ohne großen Aufwand und in akzeptabler Zeit möglich ist. Durch die Analyse der DBLP–Datenbasis stellte sich auch heraus, dass ein paar der enthaltenen Daten für eine Suche uninteressant sind und nicht in den Index aufgenommen werden brauchen, was sich positiv auf die Größe des Index auswirkt.

Die Auswertung der verschiedenen Typen von Suchabfragen erfordert teilweise spezielle Vorgehensweisen und in gewissen Fällen auch Erweiterungen der Indexstruktur, um eine effiziente Abarbeitung zu gewährleisten. Die Umsetzung dieser Vorgaben wird von der CLucene–Suchmaschine größtenteils „hinter den Kulissen“ erledigt, so dass der Anwendungsprogrammierer nur sehr wenige Klassen kennen und benutzen muss. Insb. die verschiedenen `Query`–Unterklassen spielen hier eine wichtige Rolle.

Die Bewertung der Suchergebnisse folgt bei CLucene weitgehend dem aus dem Information Retrieval sehr bekannten Vektorraum–Modell, das (einfach gesprochen) sowohl Suchabfrage als auch Dokumente als Vektoren von Termgewichten repräsentiert; Die Ähnlichkeit von Suchvektor zu Dokumentvektor bestimmt dabei die Bewertung eines Dokuments. In CLucene erledigt die Klasse `Similarity` einen Großteil der dafür anfallenden Berechnungen, wobei sie sich auf das Cosinus–Maß stützt.

Schließlich bietet die eigentliche Suchanwendung `dblpsearch` eine einfach zu bedienende und auch bei gleichzeitiger Nutzung durch mehrere Anwender leistungsfähige neue Suchmöglichkeit für das DBLP.

6.2 Erreichen der Zielvorgaben

Die am Anfang der Arbeit gesteckten Ziele konnten zum überwiegenden Teil erreicht werden. Einige der Punkte wurden im Laufe der Arbeit aufgrund von neuen Überlegungen fallengelassen. Die meisten der sonstigen nicht implementierten Vorgaben ließen sich bei ausreichender Zeit verhältnismäßig einfach ergänzen.

Die Punkte im Einzelnen:

- *Alle Daten durchsuchbar:* Die neue DBLP–Suche umfasst alle im DBLP–Datenbestand erfassten Publikationen. Bzgl. der in den Suchindex aufgenommenen Datenfelder wäre es von den Fähigkeiten der Suchmaschine ohne weiteres möglich gewesen, sämtliche Daten zu indexieren; nach Analyse der Daten wurde jedoch auf die Aufnahme einiger, im Prinzip nur für interne Verwaltungszwecke benötigter Felder verzichtet, da davon auszugehen ist, dass diese für den Nutzer kaum von Interesse sind. Sollte sich die Aufnahme dieser Felder später doch als notwendig oder sinnvoll erweisen, so ist dies ohne Probleme einzurichten.

- *Aktualität der Daten:* Die Erstellung des Suchindex ist sehr einfach möglich und erfordert nur verhältnismäßig geringe Zeit ($<$ 30 Minuten auf Testsystem). Eine tägliche (Neu-)Erstellung des Index ist somit problemlos möglich.

- *Gute Performance:* Die Leistung der neuen Suchanwendung ist, selbst bei gleichzeitiger Nutzung durch mehrere Anwender, ausreichend hoch. Die Dauer für die Auswertung, auch komplexerer Suchanfragen, liegen merklich unter denen der bisherigen Suchen.[1]

- *Ranking der Suchergebnisse:* Eine Bewertung der Suchergebnisse und entsprechende Sortierung wird von der CLucene–Suchmaschine durchgeführt. Die Bewertung der einzelnen Treffer ist auf einfache Weise direkt in der Ergebnisliste ersichtlich.

- *Schnell und einfach verfügbar:* Zum Zugriff auf die Suchanwendung wurde eine einfach Webapplikation erstellt. Diese kann ohne großen Aufwand unter jedem Webserver mit CGI–Unterstützung installiert und betrieben werden. Ein Zugriff durch die Nutzer ist damit auf einfache Weise mit jedem handelsüblichen Webbrowser über das Internet möglich. Ein Herunterladen des DBLP–Datenbestandes, die Installation zusätzlicher Programme oder eine spezielle Konfiguration auf Nutzerseite ist nicht erforderlich.

[1] Beachtet allerdings Fußnote 18 in Abschnitt 5.3.3.

- *Neue Suchtypen:* Die verwendete CLucene–Suchmaschine bietet eine Reihe weiterer Suchtypen (Ähnlichkeitssuche, Proximity- bzw. Phrasensuche, Bereichs–Suche, Suche mit Platzhaltern) an, die vollständig in der neuen DBLP–Suche verwendbar sind. Es ist davon auszugehen, dass sich insb. die Ähnlichkeitssuche, aufgrund der trotz aller Bemühungen immer noch vorhandenen unterschiedlichen Schreibweisen der Autorennamen, als nützlich erweisen wird.

  Eine Unterstützung für Synonymsuche konnte leider im Zuge dieser Arbeit aus Zeitgründen und fehlender Unterstützung in CLucene nicht implementiert werden; ebenso fehlt die Möglichkeit für Phonetische Suche, die insb. für Autorennamen vielversprechend scheint.

- *Volle Unterstützung für Boolean–Queries bzw. Kombination von Query–Typen:* Die verwendete Suchmaschine erlaubt die Kombination aller Suchtypen innerhalb einer einzelnen Abfrage. Mehrere Suchterme bzw. Einzelabfragen können problemlos mittels Boolean–Operatoren und Klammerung zu einer Gesamtabfrage kombiniert werden.

- *Kompatibilität der Anwendung:* Die Suchanwendung wurde in C^{++} entwickelt und setzt, soweit nötig, auf recht bekannten, gut unterstützten und frei verfügbaren Bibliotheken auf. Da keine speziellen Features eines bestimmten Betriebssystems oder einer Entwicklungsumgebung vorrausgesetzt werden, sollte das Bauen und Betreiben auf den meisten Plattformen ohne größere Probleme möglich sein. Getestet wurde die Anwendung allerdings nur unter Ubuntu Linux 7.04 und aufgrund bisheriger Erfahrungen ist davon auszugehen, dass bei einer Benutzung unter einem anderen System vorher zumindest gewisse Anpassungen nötig sein werden.

- *Konfigurierbare Ergebnisliste:* Aufgrund von Zweifeln an der Interessantheit einer solchen Steuerungsmöglichkeit wurde davon abgesehen, die Anzahl der anzuzeigenden Treffer konfigurierbar zu gestalten. Stattdessen wurde eine seitenweise Anzeige der Resultate implementiert, die übersichtlich und einfach zu bedienen ist und die außerdem erlaubt — wenn gewünscht — alle Ergebnisse abzurufen.

  Eine Auswahlmöglichkeit für die in der Ergebnisliste anzuzeigenden Informationen wurde (u.A. aus Zeitgründen) bisher nicht implementiert. Aufgrund der Programmarchitektur der Anwendung ist eine dahingehende Erweiterung jedoch einfach möglich.

- *Bessere Steuerung:* Es wurde ebenfalls davon abgesehen, bei der Suche eine Unterscheidung nach der Groß-/Kleinschreibung vorzunehmen, da davon

auszugehen ist, dass dies eher zu Problemen und Missverständnissen beim Nutzer führen würde. Insb. ein „Verlust“ von Treffern wäre bei der Berücksichtigung der exakten Schreibweise nicht auszuschließen, da Dokumente, die einen Suchterm mit lediglich unterschiedlicher Groß-/Kleinschreibung beinhalten, nicht zurückgeliefert werden würden.

Auch auf eine spezielle Behandlung von Sonderzeichen wurde verzichtet; Zeichen wie z.B. Umlaute in Suchtermen werden ganz normal behandelt und verglichen. Die in der bisherigen Autorensuche vorhandene Möglichkeit, bei Eingabe von Sonderzeichen auch alternative Schreibweisen des Namens zu finden, kann mittels Platzhaltern sauberer und intuitiver erreicht werden.

- *Weitere Features:* Möglichkeiten der manuellen Gewichtung von Suchtermen und die Suche mit Platzhaltern gehören zum Repertoire von CLucene und sind somit ebenfalls in der Anwendung verfügbar.

6.3 Schlussbemerkung

Leider nimmt auch in solchen Projekten wie diesem die „Alltagsarbeit“ einen breiten Raum ein. Das Aufsetzen der Entwicklungsumgebung, das Installieren von Bibliotheken und Tools, die Suche nach Informationen zu Compilern, Schnittstellen und benötigten Features der Programmiersprache und nicht zuletzt die Suche nach Fehlern — die im Normalfall meist absolut nichts mit der eigentlichen Problemstellung zu tun haben — erfordert Zeit.

Nichtsdestotrotz bietet diese Arbeit hoffentlich einen ersten Einblick in die grundlegende Problematik, die verschiedenen Aspekte sowie die spezifischeren Gegebenheiten und Methoden die bei der Entwicklung einer spezialisierten Suchanwendung eine Rolle spielen.

Insgesamt ist es interessant zu sehen, wieviel Potential für Entwicklung und Forschung bereits in solch einer, anfänglich vermeintlich einfach aussehenden, Aufgabe steckt. Selbst ohne tiefergreifende Berücksichtigung weitergehender Themen bietet ein solches Projekt viel Raum für Überlegungen und Analysen und erfordert eine intensive Beschäftigung mit den anstehenden Problemen und Sachverhalten.

Literaturverzeichnis

[ASFLfaq] **The Apache Software Foundation** Apache Lucene – FAQ. Bearbeitungsstand 25. July 2007 07:00 UTC. URL: http://wiki.apache.org/lucene-java/LuceneFAQ

[ASFLf] **The Apache Software Foundation** Apache Lucene – Features. 2006 (Erstzugriff 4. Juni 2007). URL: http://lucene.apache.org/java/docs/features.html

[ASFLiff] **The Apache Software Foundation** Apache Lucene – Index File Formats. 2006 (Erstzugriff 4. Juni 2007). URL: http://lucene.apache.org/java/docs/fileformats.html

[ASFLjd] **The Apache Software Foundation** Apache Lucene – JavaDoc. 2007 (Erstzugriff 4. Juni 2007). URL: http://lucene.zones.apache.org:8080/hudson/job/Lucene-Nightly/javadoc/index.html

[ASFLqs] **The Apache Software Foundation** Apache Lucene – Query Parser Syntax. 2006 (Erstzugriff 4. Juni 2007). URL: http://lucene.apache.org/java/docs/queryparsersyntax.html

[ASFLsc] **The Apache Software Foundation** Apache Lucene – Scoring. 2006 (Erstzugriff 4. Juni 2007). URL: http://lucene.apache.org/java/docs/scoring.html

[BR+99] **Baeza–Yates, R., Ribeiro–Neto, B., u.a.** Modern Information Retrieval. Addison Wesley Longman, 1999. Teilweise online verfügbar unter http://people.ischool.berkeley.edu/~hearst/irbook/

[BPS+06] **Bray, T., Paoli, J., Sperberg–McQueen, C.M., Maler, E., Yergeau, F.** Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, 29 September 2006. URL: http://www.w3.org/TR/2006/REC-xml-20060816

[Cpre] **Christiansen, T.** perlre – Perl regular expressions. Juni 2001. URL: http://www.perl.com/doc/manual/html/pod/perlre.html

[CCG+72] **Chapman, G., Cleese, J., Gilliam, T., Idle, E., Jones, T., Palin, M.** Monty Python's Flying Circus, Episode 35. Python (Monty) Pictures Limited, 1972.

[CLben] **Klinken, B. van** CLucene — Benchmarks. Bearbeitungsstand 6. Juli 2006, 14:33 UTC. URL: http://clucene.sourceforge.net/index.php/Benchmarks

[DFAQ] **Ley, M.** DBLP FAQ: Which software is behind DBLP? (Erstzugriff 20. März 2007). URL: http://www.informatik.uni-trier.de/~ley/db/about/faqsoft.html

[LR06] **Ley, M., Reuther P.** Maintaining an Online Bibliographical Database: The Problem of Data Quality. Präsentation EGC, Januar 2006. Online verfügbar unter http://dblp.uni-trier.de/xml/egc2006.ppt

[MRS07] **Manning, C.D., Raghavan, P., Schütze, H.** Introduction to Information Retrieval. Cambridge University Press, 2007. Online verfügbar unter http://www-csli.stanford.edu/~schuetze/information-retrieval-book.html

[NIST00] **National Institute of Standards and Technology (NIST)** IEC 60027–2, Second edition, Letter symbols to be used in electrical technology — Part 2: Telecommunications and electronics. November 2000. Online verfügbar unter http://physics.nist.gov/cuu/Units/binary.html

[R79] **Rijsbergen, C.J.** Information Retrieval. Butterworths, 1979. Online verfügbar unter http://www.dcs.gla.ac.uk/Keith/Preface.html

[Zett06] **RMIT University Search Engine Group** The Zettair Search Engine. September 2006. URL: http://www.seg.rmit.edu.au/zettair/

[R06nw] **Robertson, J.** Nine ways to fix intranet search. May 2006. URL: http://www.steptwo.com.au/papers/kmc_fixingsearch/index.html

[R05wti] **Robertson, J.** What to include in intranet search results. Juli 2005. URL: http://www.steptwo.com.au/papers/cmb_searchresults/index.html

[V06] **Vollmer, S.** Portierung des DBLP–Systems auf ein relationales Datenbanksystem und Evaluation der Performance. Diplomarbeit an der Universität Trier, März 2006. Online verfügbar unter http://dbis.uni-trier.de/Diplomanden/Vollmer/vollmer.shtml

[WMB94] **Witten, I.H., Moffat, A., Bell, T.C.** Managing Gigabytes: Compressing and indexing documents and images. Van Nostrand Reinhold, 1994.

[WPrp] **Wikipedia, Die freie Enzyklopädie** Artikel Recall und Precision. Bearbeitungsstand 3. Juni 2007, 22:15 UTC. URL: http://de.wikipedia.org/w/index.php?title=Recall_und_Precision\&oldid=32715017

[WPps] **Wikipedia contributors** Artikel Proximity search (text). Bearbeitungsstand 30. Juni 2007, 00:48 UTC. URL: http://en.wikipedia.org/w/index.php?title=Proximity_search_%28text%29\&oldid=141510878

[WPpp] **Wikipedia contributors** Artikel Perl, Abschnitt Comparative performance. Bearbeitungsstand 23. April. 2007, 21:05 UTC. URL: http://en.wikipedia.org/w/index.php?title=Perl\&oldid=125289335#Comparative_performance

[WPvsm] **Wikipedia contributors** Artikel Vector space model. Bearbeitungsstand 29. July 2007, 23:22 UTC. URL: http://en.wikipedia.org/w/index.php?title=Vector_space_model\&oldid=147944971

Anhang A

Zusatzinformationen

A.1 Die Bibliothek `libdblp`

Im Zuge der Erstellung der Programme für diese Diplomarbeit stellte sich heraus, dass ein signifikanter Teil der gleichen Funktionalität von verschiedenen Programmen benötigt wird. Um zu verhindern, dass der entsprechende Code mehrmals direkt innerhalb der Programme vorhanden ist, wurden diese Klassen in eine eigene Bibliothek `libdblp` ausgelagert.

Im Einzelnen handelt es sich um die Klassen:

FullSAXHandlerAdaptor Diese Klasse leitet sich von den in der Arabica–Bibliothek vorhandenen Klassen `SAX::EntityResolver`, `SAX::DTDHandler`, `SAX::ContentHandler`, `SAX::ErrorHandler`, `SAX::LexicalHandler` sowie `SAX::DeclHandler` ab. Sie erzeugt und konfiguriert ein Objekt vom Typ `SAX::XMLReader<std::string>`. Außerdem stellt sie (leere) Implementierungen für alle in den obigen Klassen vorhandenen Methoden bereit und ermöglicht so wiederum von ihr abgeleiteten Klassen, auf einfache Weise eine XML–Datei zu parsen und dabei auf beliebige SAX–Ereignisse zu reagieren, indem diese die gewünschten Methoden mit eigenen Implementierungen überschreiben.

Field Diese Klasse verwaltet ein (Schlüssel,Wert)–Paar und dient zur Speicherung der Daten eines Datenfeld–Elements aus dem DBLP–Datenbestand.

Publication Diese Klasse repräsentiert einen kompletten Publikationsdatensatz im DBLP. Neben Werten für `Name`, `Key` und `MDate`, die die entsprechenden Attributswerte bzw. den Elementnamen aus der DBLP–XML–Datei beinhalten, verwaltet sie auch eine Liste mit allen dem Publikationen–Datensatz zugehörigen Datenfeldern (in Form von `Field`–Objekten).

DBLPParser Diese Klasse leitet sich von `FullSAXHandlerAdaptor` ab und stellt einen speziell auf die Datei `dblp.xml` ausgerichteten Parser dar. Sie filtert bzw. interpretiert die Low–Level–SAX–Ereignisse wie „Element startet“ oder „Zeichen gefunden“ und generiert daraus neue, übergeordnete Ereignisse. Abgeleitete Klassen können damit auf die Ereignisse `startDBLP` (Das Root–Element `<dblp>` wurde geöffnet, Start der Datei), `field` (ein komplettes Feld–Datenelement mit Inhalt wurde eingelesen), `publication` (Ein kompletter Publikationen–Datensatz mit allen Datenfeldern wurde eingelesen) und `endDBLP` (Das Root–Element `<dblp>` wurde geschlossen, Ende der Datei) reagieren.

StringTools Diese Klasse stellt für C++–`std::string`–Objekte einige Methoden bereit, die offensichtlich nicht in den Standardklassen bzw. Bibliotheken vorhanden sind. Es sind das eine Methode zum Suchen und Ersetzen einer Zeichenkette durch eine andere, eine Methode zum Suchen und Ersetzen von einzelnen Zeichen durch andere und eine Methode zum Maskieren von speziellen Zeichen für die Ausgabe in HTML.

TCharString Diese Klasse kapselt eine Zeichenkette; die eigentliche Aufgabe ist die Konvertierung von und nach den von CLucene benutzten `TCHAR`–Zeichenketten. Bei `TCHAR` kann es sich, je nachdem wie CLucene kompiliert wurde (mit _UCS2 oder mit _ASCII), um einfache `char`- oder um breite `wchar_t`–Daten handeln. Diese Klasse erlaubt die einfache Konvertierung in beide Richtungen und verwaltet außerdem automatisch den erforderlichen Speicher. Außerdem übernimmt sie, nach Wunsch, eine Konvertierung von UTF–8–Zeichen.

Wie bereits im Laufe dieser Arbeit angesprochen, handelt es sich bei der Zeichenkonvertierung nicht unbedingt um ein triviales Thema und Verbesserungen hier sind sicherlich möglich. Nichtsdestotrotz sollte diese Klasse die Arbeit mit den entsprechenden Zeichenketten bereits um einiges vereinfachen.

Neben diesen Klassen stellt die Bibliothek noch die zwei Include–Dateien `config.h` und `stl_ns.h` mit ein paar mehrfach gebrauchten Konstanten zur Verfügung.

Der Quellcode der Bibliothek ist ebenfalls auf der beiliegenden CD bzw. unter http://diplomarbeit.c2226.de/ erhältlich. Genauere Informationen über die bereitgestellten Klassen und Methoden finden sich dort.

A.2 Daten des Test- und Entwicklungssystems

Tabelle A.1: Hardware des Test- und Entwicklungssystems

System	Asus AB-P 2800 Book Size Barebone
Prozessor	Intel(R) Pentium(R) 4 CPU 2.80GHz
Mainboard	ASUS P4R8L
Arbeitsspeicher	2 GiB DDR2–RAM
Festplatte	Hitachi HDS728080PLA380, 82.3GB Hard Drive SATA300 Model, 7677 KiB Cache

Tabelle A.2: Verwendetes Betriebssystem, Software und Bibliotheken

Betriebssystem	Kubuntu Linux 7.04
Kernelversion	Kernel 2.6.20-16
Compiler	gcc (GCC) 4.1.2 (Ubuntu 4.1.2-0ubuntu4)
Linker, etc.	binutils 2.17.20070103cvs-0ubuntu2
Make	GNU Make 3.81(-3build1)
libc	libc 2.5-0ubuntu14
Arabica	arabica-Jan2007 (arabica-Jan2007.tar.bz2)
CLucene	CLucene 0.9.18 (clucene-core-0.9.18.tar.bz2 + clucene-contrib-0.9.16a.tar.bz2)
Lucene	Lucene 2.1.0 (lucene-2.1.0-src.tar.gz)
MG	mg-1.3.64x.1 (mg-1.3.64x.1.tar.bz2)
WordNet	WordNet 3.0 (WordNet-3.0.tar.bz2)
Zettair	zettair-0.9.3 (zettair-0.9.3.tar.bz2)
Diagramme erstellt mit	gnuplot 4.0

A.3 Erfasste Datenfelder

Tabelle A.3 stellt eine Übersicht über die mittels des Programms `dblp2index` erfassten bzw. erzeugten Datenfelder im Suchindex dar. Angegeben sind jeweils die bei der Erstellung bzw. Aufnahme verwendeten Einstellungen. Dies umfasst ob die Felder indexiert wurden (und damit durchsuchbar sind), gespeichert wurden (d.h. mit den zurückgelieferten Treffern direkt abfragbar sind) und ob ihr Inhalt durch die Klasse `StandardAnalyzer` behandelt und in einzelne Token aufgeteilt wurde.

Tabelle A.3: Die von `dblp2index` erfassten Datenfelder

Feldname	Indexiert?	Gespeichert?	Tokenized?	Anmerkung
address	X	-	X	
author	X	X	X	
booktitle	X	-	X	
cdrom	-	-	-	
chapter	X	-	-	
cite	X	-	-	
crossref	X	-	-	
editor	X	-	X	
ee	-	X	-	
hasEE	X	-	-	Neu erzeugtes Feld
isbn	X	-	-	
journal	X	-	X	
key	X	X	-	Aus key–Attribut
mdate	X	-	-	Aus mdate–Attribut
month	X	-	-	
note	X	-	X	
number	X	-	X	
pages	X	-	-	
pubType	X	-	-	Aus Elementnamen
publisher	X	-	X	
school	X	-	X	
series	X	-	X	
title	X	X	X	
url	-	-	-	
volume	X	-	-	
year	X	X	-	

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde auch nicht veröffentlicht.

Trier, den 20. August 2007

Thorsten Thielen

(in astounded voice)
„Lemmon curry?“